Efficient Clustering Algorithms for Large-scale Graphs

著者	塩川 浩昭
year	2015
その他のタイトル	大規模グラフに対するクラスタリングの高速化に関する研究
学位授与大学	筑波大学 (University of Tsukuba)
学位授与年度	2014
報告番号	12102甲第7288号
URL	http://hdl.handle.net/2241/00129427

Efficient Clustering Algorithms for Large-scale Graphs

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2015

Hiroaki Shiokawa

ABSTRACT

Recent advances in social and information science have provided linked data that model our society and the natural world around us. Since graphs are fundamental data structures that can naturally represent data entities as well as the relationships among entities, their importance is increasing in representing complicated structures and schema-less data such as data generated by Twitter, Facebook and various complex networks. For these complex networks, graph cluster analysis is one of the most important techniques in various research areas such as data mining, social science and marketing.

The problem of the graph cluster analysis is to find clusters of nodes which are densely connected within the cluster and sparsely connected among clusters, and this problem has been studied for some decades in many fields. Since the size of graphs is increasing significantly in the big data era, efficient clustering algorithms that can handle large-scale graphs are highly demanded.

In order to achieve efficient clustering of large-scale graphs, this dissertation investigates three novel algorithms taking statistical properties of real-world graphs into consideration: (1) we propose an incremental agglomerative graph clustering algorithm for modularity-based clustering, (2) we investigate a data parallel clustering method for modularity-based graph clustering by using SIMD instructions, and (3) we introduce a novel clustering algorithm for structural clustering by handling two-hop away nodes in the real-world graphs for efficient clustering. As a result, in the experiments over real-world and synthetic datasets, this dissertation proves that the algorithms can achieve not only efficient clustering but also highly accurate clustering results for large-scale graphs. By providing efficient and sophisticated approaches that suit for large-scale graphs, the algorithms will help in increasing the effectiveness of clustering in a wide range of applications.

Acknowledgements

First of all, I am sincerely and extremely grateful to my supervisor, Professor Hiroyuki Kitagawa who opened the gate for me to pursue Ph.D degree. This thesis has been done under the direction of him, and would not have been possible without his helpful advises and encouragement. I would like also to thank great faculty members, Associate Professor Toshiyuki Amagasa, Assistant Professor Yasuhiro Hayase, and Assistant Professor Chiemi Watanabe. They greatly helped and encouraged me.

I am very grateful to my doctoral committee, Professor Hiroyuki Kitagawa, Professor Tetsuya Sakurai, Professor Koichi Wada, Professor Sadaaki Miyamoto, and Associate Professor Toshiyuki Amagasa for their valuable suggestions and constructive recommendations. Their efforts to strengthen and improve my thesis from different technical angles are wholeheartedly appreciated.

Also, my thesis was highly supported by so many my colleagues of Nippon Telegraph and Telephone (NTT) Corporation. I would like to thank the members of Distributed Processing Technology Project at NTT Software Innovation Center, Dr. Makoto Onizuka (he is now a professor at Osaka University), Dr. Yasuhiro Fujiwara, Mr. Takeshi Yamamuro, Mr. Junya Arai, Mr. Yasutoshi Ida, Dr. Machiko Toyoda, Dr. Takeshi Mishima, Mr. Yasuhiro Iida, Dr. Toshimori Honjo, Dr. Sotetsu Iwamura and Mr. Seiji Kihara. I got the idea of this work thanks to the helpful advises from them. I would especially like to thank Dr. Makoto Onizuka and Dr. Yasuhiro Fujiwara for their supports and advises. I learned visions and philosophy of research from them.

I would also like to thank members of Kitagawa Data Engineering Laboratory. I am grateful to my seniors and friends, Dr. Hiroyuki Toda, Dr. Yutaka Kabutoya, Dr. Tsubasa Takahashi, Dr. Yuto Yamaguchi, Dr. Salman Ahmed Shaikh and Mr. Takahiro Komamizu for their support and good will. I enjoyed many discussions on research, their sharing ideas, and collaboration of works as well as social events. Ms. Hiroko Odagiri, Ms. Yumiko Hisamatsu and Ms. Tetsuko Sato also helped me to accomplish this work.

I would like to thank my parents, Masahiro and Yuko, my younger brother and sister, Yuki and Ayaka, and my grandparents, Yoshihiro and Yasuko, for always being supportive. Without your comprehensive supports, I would not be here.

Finally, I would like to thank my wife, Riko, for standing beside me. She took my long journey to be always in my side to give her absolute support, love, warmth and understanding for the success of my study. Your love and encouragement have always helped me.

Hiroaki Shiokawa
January 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in technology and computing power now provide the possibility of collecting and mining a massive amount of real-world data. Mining such data will allow us to understand the structure and the workings of real systems and to find unknown and interesting patterns.

Many types of real-world datasets can be modeled with *graphs*. Graphs can represent data entities as well as the relationships among entities and provide a powerful mathematical tool to represent the relations in the data. They arise in a wide range of application domains from social networks to biological networks and beyond. For example, a social network is a graph connecting people who contact or interact with each other; nodes and edges represent people and interactions among people, respectively. Social networks are not limited to "online social networks" such as Facebook, Twitter and LinkedIn. Other examples of social networks are networks of people collaborating with each other, co-authorships and co-appearance, as well as networks of communication among people such as telecommunications and emails. An information network is a graph of entities such as World Wide Web (WWW), network of citations, and word co-occurrence networks (*a.k.a.* wordnet). Technical networks refers to manmade graphs such as the Internet topology, the electric power grid, road networks, railways and airline routes. Biological networks represent biological systems such as networks of metabolic pathways, protein-protein interactions, the food web, and the network of blood vessels.

Traditionally, graphs are modeled as random graphs [1]. However, empirical studies on real-world graphs have revealed that real-world graphs are organized into

topological structures of some intrinsic properties [2–5]. Hence, it is important to uncover the intrinsic structures hidden in the real-world graphs to advance scientific and industrial applications.

In order to understand the structure and the functions hidden in such real-world graphs, a great many graph mining techniques such as PageRank [6, 7], shortest path discovery [8, 9], influence maximization [10, 11] and frequent pattern discovery [12, 13] have been developed in the last two decades. An extremely well-studied structural property of real-world graphs is their cluster structure. The cluster structure captures the tendency of similar nodes in the graph to group together into clusters. This property has been observed in many real-world graphs [2, 3]. In order to understand the structures and the functions hidden in massive scale graphs, graph cluster analysis (*a.k.a.* community detection) is one of the most important techniques in various research areas such as data mining [14], social science [15], computer networks [16]. A cluster can be regarded as a group of nodes that are densely connected within a group while only sparse connections between different groups. By discovering the hidden cluster structures in large-scale graphs we can understand the characteristics and interrelationships of the nodes forming the graph. Here, we give some real-world applications of graph cluster analyses.

Community Detection in Social Networks: One of the most popular applications of graph clustering is social analysis. As we described before, graphs can represent human relationships such as friendship, co-authorship, telephone calls, and emails in the form of the social networks. Graph clustering algorithms naturally fit to community detection in massive graphs. The communities have natural interpretations in the context of a variety of social networks:

- For the case of friendship networks such as Facebook, Twitter and LinkedIn, communities stand for groups of members who may know each other and may, therefore, be linked with one another. This is useful in determining important associations in the underlying friendships. In addition, blogging communities often behave like social networks, and contain links among related blogs. Graph cluster analyses are also useful for determining the closely related blogs that are likely to cover the same topics.
- An interesting application in the context of the Enron scandal was to determine important email interactions among groups of Enron employees. Graph clustering algorithms are very useful in order to isolate dense email interactions among different groups of customers. This approach can be used for a

variety of intelligence applications such as determining suspicious communities through the grouping of interactions.

Cluster Analyses on Biological Networks: In recent years, large amounts of biological networks have been generated in the field of medical care and bioinfomatics. Some biological networks are naturally decomposed into some components, which are commonly referred to as modular networks such as transcriptional modules, protein complexes, gene functional groups, and signaling pathways. For this reason, graph clustering methods are often used for the following biological networks:

- Recently, large amounts of mammalian protein-protein interaction (PPI) networks have been generated and are available for public access (*i.e.* BioGRID [17], HPRD [18] and MINT [19]). PPI networks refer to physical contacts established among proteins as a result of biochemical events and/or electrostatic forces; each node represents a protein/gene and each edge represents a physical contact between proteins/genes. From a biological perspective, protein and gene interactions imply the key mechanisms related with disease and health. In fact, a lot of existing studies have reported that extremely important mechanisms can be uncovered through graph cluster analyses [20–23].
- The description of structural and functional connectivity in the human brain has attracted considerable attention [24]. The human brain is structurally and functionally organized into a complex network enabling segmentation and integration of information processing. Recent studies have suggested that a combination of MRI techniques together with graph cluster analyses can help us to map structural and functional connectivity patterns of the human brain [25–28].

Graph cluster analyses are also used in many other applications detailed in Chapter 2. As we can see above, the graph cluster analyses are undoubtedly significant for graph data mining and its application.

1.1 Motivations

As described above, graph cluster analyses are useful in a wide range of applications. In order to find clusters from real-world graphs, a lot of graph clustering algorithms have been proposed over the last few decades in many fields, particularity in computer science and physics. The main challenge of graph clustering algorithms is to find clusters, sets of nodes which are densely connected within clusters and sparsely connected among clusters. Existing clustering methods fail to scale well and so suffer from high computation cost. This is because the size of the above real-world graphs are very large. For example, the most well known graph is the WWW, which now contains more than 50 billion web pages and more than one trillion unique URLs [29]. In addition, a recent snapshot of the friendship network of Facebook contained over 864 million daily active users [30]. Furthermore, LinkedData is also going through exponential growth, and it now consists of 31 billion RDF triples and 504 million RDF links [31]. Even though the size of these graphs continues to increase, most of existing graph clustering algorithms require exhaustive computations that repeatedly evaluate all nodes and edges. Therefore, it is difficult to apply current algorithms to large-scale graphs.

Graph cluster analyses of large-scale graphs have applications in many domains. They include recommendation and marketing where clustering speed is of prime importance. In addition, the above application domains also require highly accurate clustering results. Therefore, this thesis presents efficient graph clustering algorithms that maintain their clustering quality even for large-scale graphs. The main objectives of this research are as follows:

- **Efficiency:** The algorithms proposed here are designed to achieve faster clustering speeds than state-of-the-art alternatives for large-scale graphs, i.e. those with more than a few million nodes.
- **Accuracy:** The algorithms are designed to match the clustering quality of state-of-the-art algorithms for large-scale graphs.

As stated before, even though the state-of-the-art algorithms can produce clusters of good quality, it is difficult to apply them to large-scale graphs due to their clustering speed limitations. This thesis provides efficient algorithms that are applicable to large-scale graphs and produce clusters of high quality. Our proposals will help in increasing effectiveness of clustering in a wider range of applications.

1.2 Contributions

The goal of our research is to present efficient, accurate and scalable graph clustering algorithms for extremely large-scale graphs. This thesis presents novel algorithms that overcome the low efficiency of the current state-of-the-art graph clustering methods for large-scale graphs based on interesting structural properties of real-world graphs. Compared to state-of-the-art algorithms, the proposals can efficiently analyze graphs at least with more than several million nodes and edges. Moreover, this thesis reveals that the proposals also produce high accurate clustering results, matching those of existing methods. The proposed algorithms can find well-clustered results from large-scale graphs within short computation time. In this section, we first give the main ideas underlying our contributions and then give summaries of our proposed graph clustering algorithms.

1.2.1 Main Ideas: Structural Properties of Graphs

The basic concept of this thesis is to utilize the structural properties of real-world graphs for improving clustering speed. Historically, a great deal of work has revealed that real-world graphs have several structural properties [32]. One of the most famous properties is the "small-world effect" [3], also known as "six degrees of separation" [2], and the scale-free behavior of graphs [4,5]. It is known that most of real-world graphs that exhibit the small-world effect has two robust measures: *high clustering coefficient* and *power-law degree distribution*.

The clustering coefficient is a measure of the degree in which nodes in a graph tend to be in the same cluster. If the value of the clustering coefficient of a graph is high, nodes of the graph tend to create densely connected groups [33]. In fact, Watts and Strogatz revealed that many real-world graphs have significantly higher clustering coefficients than random graphs [3]. This means that in clusters of real-world graphs most adjacent nodes are densely connected to each other and they are likely to share large portions of neighboring nodes.

Another measure of the structure of real-world graphs is degree distribution, which characterizes the distribution of node degrees. It has been shown that real-world graphs tend to have degree distributions that follow the power-law [4, 5]. This means that most nodes have relatively small degrees while just a few nodes have significantly larger degrees.

These properties indicate that real-world graphs are fundamentally different from random graphs [1]. This thesis introduces three graph clustering algorithms that are designed to utilize the above structural properties for computational efficiency. The algorithms are summarized in the following subsections.

1.2.2 Fast Algorithm for Modularity-based Graph Clustering

This proposal offers high efficiency for modularity-based graph clustering. Recently, the modularity-based graph clustering algorithm of [34] has become a de facto graph clustering tool and it is widely used in various applications. The basic concept of modularity-based graph clustering is to define the subgraphs as clusters that have significantly different topological structures from random graph structures. Since finding the clusters that maximize the modularity is an NP-complete problem, many greedy approaches have been proposed [35–38]. However, these methods including the state-of-the-art algorithm suffer from the following three bottlenecks when applied to large-scale graphs: (1) they exhaustively and iteratively process all nodes and all edges in the large-scale graph to find clusters, (2) the random node selection employed by the algorithm involves unnecessary computations for finding clusters, and (3) its random node accesses increase the cost of node referencing. From these reasons, existing algorithms incur high computation costs for clustering.

To overcome the above clustering speed limitations, we propose a greedy algorithm named *Incremental Modularity Agglomeration Clustering (IMAC)*. IMAC focuses on the structural properties of real-world graphs: *high clustering coefficients* and *power-law degree distributions*. Since IMAC considers both structural properties, it avoids the exhaustive computation of all nodes and edges. Specifically, IMAC employs the following three approaches for efficient clustering: (1) it incrementally aggregates nodes, which are placed in the same cluster, into a single vertex, (2) it incrementally prunes modularity gain computations of nodes that have already been clustered, and (3) it optimizes the order of vertex selection for efficient clustering.

As a result, IMAC has three advantages. The first is its high efficiency. IMAC is considerably faster than existing approaches such as CNM [36] and BGLL [38]. Specifically, our experiments show that IMAC computes clusters from large-scale graphs of more than 100 million nodes within 3 minutes. The second is its high accuracy. Our approach provides clustering results with high modularity; it returns

almost the same modularity scores as the state of the art approach, BGLL. The third advantage is its effectiveness for real-world graphs. As described in Section 1.2.1, IMAC is designed to effectively utilize the structural properties of real-world graphs such as the clustering coefficient and power-law degree distribution. For this reason, IMAC offers excellent clustering speed for large-scale complex networks. We detail this algorithm in Chapter 3.

1.2.3 Vectorized Algorithm for Modularity-based Clustering

This proposal is a vectorized modularity-based graph clustering algorithm, named *ParBGLL*, that uses SIMD instructions. As described in the previous section, existing modularity-based graph clustering algorithms, including the state-of-the-art method, suffer from low efficiency due to their exhaustive computations and randomized data accesses.

Our proposal consists of two building blocks. The first, in order to reduce computation cost of node referencing, we adopt a CPU cache efficient graph data representation. Specifically, we employ *compressed row storage* (CRS format for short) [39] instead of the adjacency list representation. CRS format, one of the most popular storage formats for sparse matrixes, puts the subsequent nonzero elements of the sparse matrix rows in contiguous memory locations. This increases the efficiency of node referencing. Our algorithm extends the CRS format to cover both adjacency lists and degrees, and increases the cache hit ratio for efficient clustering. The second, in order to reduce the time taken to compute modularity gain, we employ a data parallel method with Streaming SIMD Extensions (SSE), which is the SIMD instruction set extension of the x86. SIMD instructions are extended instructions present in most modern CPU designs in order to improve the performance of multimedia applications. They perform the same operation on multiple data points simultaneously, and exploit the data level parallelism of the CPU core but not concurrency. As described above, real-world graphs have highly skewed degree distributions and there are many small degree nodes. Hence, by using SIMD instructions to compute these small degree nodes, we improve the clustering speed for large-scale graphs. In addition to the above SIMD-based modularity computation, we propose an efficient modularity computation form to reduce the number of instructions that are required for each modularity gain computation by transforming the modularity's formula into simple representations.

These approaches have three major advantages. The first is that only small com-

putation cost is needed to extract clusters from large-scale graphs. Our approach successfully overcomes the bottlenecks of the state-of-the-art method BGLL; the extended CRS format increases the cache hit ratio and the data parallel method with SIMD instructions performs the modularity gain computations efficiently. The second advantage is that our proposal has better scalability than the state-of-the-art approach BGLL. The clustering speed of our approach increases almost linearly with the parallelism level for large-scale graphs. The third advantage is the high modularity of the clustering results. Our evaluation reveals that the clustering results produced by our approach have almost the same modularity scores as the state-of-the-art method BGLL. We detail the algorithm in Chapter 4

1.2.4 Fast Algorithm for Structural Clustering Algorithm

Although modularity-based approaches are effective for many applications, recent research has pointed out that they fail to fully reproduce the ground-truth from large-scale graphs [40]. In order to overcome the above drawback of modularity-based methods, density-based algorithms have been developed recently [41]. The density-based methods evaluate the density of edge connections between adjacent nodes, and they regards extremely densely connected subgraphs as clusters. It is reported that density-based methods offer better clustering results than modularity-based equivalents, however, their computation cost is significantly higher.

To improve the efficiency of density-based approaches, we propose a novel algorithm based on a basic property of real-world graphs: *a node and its two hop away nodes share a lot of neighbors*. By utilizing this property for clustering, our algorithm successfully reduces the number of edges evaluated in the clusters.

To the best of our knowledge, our proposal is the first solution to achieve both high efficiency and highly accurate clustering results at the same time for large-scale graphs. Experiments confirm that it outperforms the existing methods in terms of running time without sacrificing clustering quality. Even though existing algorithms do offer enhancements in application quality, they are difficult to apply to large-scale graphs due to their clustering speed limitation. However, by providing sophisticated approaches that suit large-scale graphs, the proposed method will help to improve the effectiveness of a wide range of applications. We detail this algorithm in Chapter 5.

1.3 Overview of the Thesis

In the above sections we have presented the background and motivations of this work, and briefly presented the main contributions of this thesis. In this section, we detail the structure of this thesis:

Chapter 2: In this chapter, we overview existing studies on graph clustering algorithms such as *graph partitioning algorithms*, *modularity-based algorithms* and *structural clustering algorithms*. Specifically, we focus on the advantages and disadvantages of existing algorithms since they employ diverse definitions of graph clusters. Besides discussing existing algorithms, we also introduce applications that use graph clustering.

Chapter 3: First, in Chapter 3, we address the goal of finding clusters based on *modularity* from extremely large-scale graphs (i.e. greater than 100 million nodes); we propose *incremental modularity agglomerative clustering (IMAC)* which utilizes clustering coefficient and power-law degree distribution of graphs for efficient clustering. We conduct evaluations to confirm the effectiveness of IMAC by using both real-world datasets and synthetic datasets.

Chapter 4: Then, in Chapter 4, we propose the fast modularity-based clustering approach by using data parallel computation schemes. In order to raise the computation efficiency of data parallel schemes, we vectorize the state-of-the-art modularity clustering algorithm BGLL. We evaluate this vectorized approach by using both real-world datasets and synthetic datasets in Chapter 3.

Chapter 5: Last, in Chapter 5, we propose a novel structural clustering algorithm, named *SCAN++*, for finding clusters defined by *structural similarity* from large-scale graphs. In order to achieve efficient clustering, we focus on the clustering coefficient of real-world graphs. SCAN++ makes effective use of the clustering coefficient to reduce the computation cost of clustering. Experiments on both real-world datasets and synthetic datasets verify the efficiency and effectiveness of SCAN++.

Chapter 6: In the final chapter, we provide the main conclusions of this thesis and outline some directions for future advances.

Chapter 2

Background and Survey

Various kinds of graph cluster analyses are now possible with the emergence of real-world graphs. This chapter outlines the basics of graph clustering, its applications and approaches.

2.1 Graph Clustering Applications

Mining large-scale real-world graphs has many different applications such as understanding the function of a system, and modeling and predicting its behavior. In this section we review some of the key application areas of graph clustering such as information networks, social networks, biological networks, and database systems.

2.1.1 Information Networks

In any information network, graph clustering serves as a tool for analyzing, modeling, and predicting the function, usage and evolution of the network. Applications include business analysis, marketing, infrastructure improvements, and identifying anomalous use.

In computer networks, clustering may be used to identify relevant substructures and to analyze the connectivity for the purposes of modeling and structural

optimization. One representative example is the topology design of the Internet. Grout and Cunningham used a graph clustering approach for optimizing the network topology in order to reduce the total network traffic load [42, 43]. They modeled the network topology, which consists of network switches and non-switches (*i.e.* personal computers and servers), as a graph, and then detected the sub-topology containing high network traffic loads by using graph clustering. In the World Wide Web, by representing each web page as a node and each hyperlink as an edge and then subjecting the hypertext documents to graph clustering, we can identify topics and other entities formed by several interconnected documents [44,45]. Those studies organized hypertext documents by using graph clustering, and then estimated the topics of all clusters.

Graph clustering is also used in the structural design and operation of wireless ad-hoc networks [46–49] and sensor networks [50]. Given that such networks exhibit dynamic topology changes, graph clustering is useful to find better network routing. For example, Nguyen *et al.* and Dinh *et al.* used graph clustering for mobile ad-hoc networks (MANET) [48, 49]. By extracting clusters in the MANET, they directly route or forward messages (*i.e.* network packets) to nodes in the same cluster as the destinations. By doing this way, they avoid unnecessary messages forwarded through nodes in different clusters, which reduces the number of duplicate messages as well as the overhead information.

2.1.2 Social Networks

Applications of graph clustering in social networks include identifying groups of individuals "exposed" to the influence of a certain individual of interest, such as identifying terrorist networks when a member is known or locating potentially infected people when an infected and contagious individual is discovered. Graph clustering of a social network also helps to identify mechanisms such as the formation of trends and behavior [51–56].

Zhou *et al.* applied the graph clustering approach to detect communities from social message networks such as emails, tweets on Twitter, and Facebook updates [52]. In order to uncover the communities of each user as well as their associated topics and communities, they designed a latent community model called COCOMP. In the work [52], they applied their approach to the tweets collected from United States President Barak Obama's account from November 2009 to February 2010. They revealed that the President roughly belongs to the following five communities.

President: comments on his advocacy of the Presidency, Public Policy: related to domestic and international politics and policy, Holidays: mainly about wishing the best for American families and friends, Senate Vote: related to health bills and Blessing Haiti: a response to the tremendous earthquake.

Finding influential individuals from users of social networks and micro-blogging services is one of the most critical issues for applications such as viral marketing. This problem is formally defined as the problem to find the individuals that influence the greatest number of users. The approach proposed by Chen *et al.* offers the effective detection of influential individuals based on SCAN [54]. In order to generate a small set of candidate individuals, this approach extracts clusters, hubs and outliers from graphs by their SCAN-based algorithm. Then, it identifies the influential individuals from just the significantly large clusters and hubs that connect a lot of clusters. As a result, they revealed that information can spread easily from hubs to many adjacent clusters; their approach outperforms the previous approaches [57] for real-world social networks.

Pei *et al.* utilized the online micro-blogging service Twitter to track the dynamically changing topics that are discussed on Twitter [55]. Since such online micro-blogging services are noisy, informal and surge quickly, they investigated an incremental density-based clustering approach. By finding clusters from Twitter and using Part-of-speech Tagger [58] for topic identification, their method can effectively track, on the fly, events and topic dynamics even from large volumes of social network data. For example, their approach detected the emergence in Twitter of the "SOPA (Stop Online Piracy Act) protest" in January 2012.

In the current information society, the study of social networks tends to overlap the study of information networks, as the popularity and significance of electronic messaging has become overwhelming in the big data era.

2.1.3 Biological Networks

In the field of bioinfomatics, graph clustering tasks typically deal with the classification of gene expression data such as gene-activation dependencies [59, 60]. Xu *et al.* [59] and Boyer *et al.* [60] used graph clustering approaches for genome analysis. As we described in the previous chapter, gene-activation dependencies can be represented by graphs. By using graph clustering approaches to identify gene-activation dependencies, we can extract groups of strongly interacting genes that may have

some helpful functions. For example, Xu *et al.* applied graph clustering to a gene expression data set of Arabidopsis, they found a known *cis-acting element* of chitin responsive genes from the dataset [59].

As well as genes, protein and gene interactions encode the key mechanisms determining disease and health [20–23, 61]. In fact, such mechanisms can be uncovered through graph analyses. For example, in order to effectively identify the key functional modules and biomarkers underlying the mechanisms of disease and toxicity, Ding *et al.* applied graph clustering to PPI networks [61]. They reported that SCAN [41] could find helpful biomarkers in real-world datasets. For example, Ding *et al.* applied SCAN to the gene interaction data of breast cancer patients. The clusters, hubs and outliers obtained by graph clustering were better prognostic biomarkers than the traditional biomakers; the former were more helpful for identifying the patients not needing additional chemotherapy.

Another biological application of clustering is epidemic spreading [62,63]. Newman [63] studied SIR-type epidemic processes in a special class of graphs and found that graphs with a cluster structure have smaller epidemics, but a lower epidemic threshold, making it easier for diseases to spread.

2.1.4 Database Systems and Distributed Systems

When storing a large set of data, a key question is how to group the data into pages in physical memory. A single page is typically large enough to contain multiple elements, only a small fraction of the entire dataset. Therefore, it is natural choice to partition the relevant data, which might be retrieved by the same query, into the same group.

Diwan *et al.* proposed a paging method that applies clustering to tree-like data [64]. Wu *et al.* [65] and Agrawal *et al.* [66] addressed the graph-based solutions for the database organization issue. They provide graph clustering-based data layout solutions in order to improve query speed for searching nodes and computing shortest paths, and implementing distance queries for scale-free graphs. Bradley *et al.* used a k-means like algorithm to partition a large database in one scan by using a limited memory buffer [67]. Recently, Wu *et al.* presented *SemStore* [68], a semantic-preserving distributed RDF triple storage scheme for scalable SPARQL query processing. They designed an RDF partition algorithm in order to reduce query processing time.

As well as database systems, distributed systems can also utilize graph clustering for load balancing. A standard approach for scalable computation is to partition the input graph into smaller units which are then processed by a large distributed system such as GraphLab [69], Trinity [70] and Pregel [71]. For these distributed systems, partition skewing may degrade scalability, so finding a balanced graph partition is an important task for load balancing. The problem of finding a balanced graph partition, one that minimizes the number of cut edges, is known as the *balanced graph partitioning* problem; it has been studied extensively over the last two decades. To support graphs with million of nodes, multi-level partitioning solutions, such as Metis [72] and Scotch [73], were proposed.

2.1.5 Other Applications

Beyond the above applications, graph clustering is used in many other application domains.

User Recommendation

Social tagging systems have emerged as a popular way for users to annotate, and share resources on the Web, such as Yahoo! Delicious, and Flickr. However, due to the extreme popularity of such systems, a user is easily overwhelmed by the large amount of data and it is very difficult to dig out the information that he/she is interested in. The modularity-based graph clustering approach suggested by Zhou *et al.* [74], can help users to discover other users with common interests automatically and effectively. It obtains an undirected weighted tag-graph for each user. In each graph, nodes represent the tags used by each user, and edges represent co-occurrences of tag pairs. It then extracts clusters as the topics that represent a user's interests by using a modularity-based clustering algorithm [36] in each graph. Finally, it computes the interest-based similarity among users by measuring the KL–divergence [75] of the topics for each user. Their approach can discover users with common interests more effectively than the memory-based [76] and model-based [77] algorithms.

Event Detection

Recent micro-blogging services such as Twitter allow users to continuously report their real life events. Detecting life events would be useful for understanding what users are really thinking and doing. Therefore, event detection algorithms have long been a research topic [78, 79]. Weng *et al.* applied modularity-based clustering to wavelet-based signals for event detection [80]. They build signals computed by wavelet analysis for individual words, which capture only the bursts in the words' appearances. They then obtain a graph whose nodes and edges represent signals and correlation values between signals, respectively. Finally, they detect events by using a modularity-based algorithm. Their modularity-based approach shows better performance than the LDA-based approach [81] in finding events.

Landmarks Detection

Social photo sharing applications such as Flickr have emerged as a popular way to share and annotate photos by using tags. However, they suffer from the limitations imposed by tags such as polysemy, lack of uniformity and spam. Thus it is difficult for users to find photos of interest from the photo collections organized by the tags. In order to achieve efficient photo searches of tagged photos, Papadopoulos *et al.* proposed a landmark detection method based on SCAN [82]. Landmarks are the representative objects that are frequently found in photos, and the clustering of tagged photos is regarded as the detection of popular landmarks. They first build a similarity image graph from tagged images by computing the cosine similarity of SIFT descriptors [83] and the co-occurrences of tags between photos. They then utilize SCAN to identify clusters, hubs and outliers from the similarity image graph; the clusters are used for detecting landmarks in the photo collections, and the hubs and the outliers are removed from the collections as noises. They estimate landmarks by ranking based on the appearance frequency of the tags included in the cluster. Their approach discovers landmarks more effectively than the k-means based algorithm [82].

2.2 Graph Clustering Algorithm

Due to the importance of graph clustering in many application domains, many graph clustering algorithms have been proposed over the last few decades. In this section, we first define the problem addressed by graph clustering algorithms in Section 2.2.1, which we address in this thesis, and then we survey the graph clustering algorithms in Sections 2.2.2 to 2.2.4.

2.2.1 Problem Definition

A cluster can be regarded as the group of nodes that are densely connected within a group and sparsely connected to different groups. Generally, graph clustering algorithms can be formalized as follows:

Problem 1 (Graph clustering)

Given: *Graph* $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$*, where* $\mathbb{V}$ *and* $\mathbb{E}$ *are sets of nodes and edges included in the graph* $\mathbb{G}$*, respectively.*

Find: *All disjoint clusters* $\mathbb{C}_i = \{\mathbb{V}_i, \mathbb{E}_i\}$ *where* $\mathbb{V}_i \subseteq \mathbb{V}$ *and* $\mathbb{E}_i \subseteq \mathbb{E}$*.*

In this thesis, to simplify the representations, we assume graphs are undirected and unweighted. Moreover, clusters do not have nodes overlapping with those of other clusters, i.e., $\mathbb{V} = \sum_i \mathbb{V}_i$ and $\mathbb{V}_i \cap \mathbb{V}_j = \emptyset$ for any $i \neq j$. In addition, we denote a set of clusters as $\mathbb{C} = \{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_k\}$ when a graph $\mathbb{G}$ has k clusters.

Based on the problem definition given by Problem 1, this section discusses and surveys the graph clustering algorithms categorized as follows:

- Graph partitioning (Section 2.2.2)
- Modularity-based clustering (Section 2.2.3)
- Structural clustering (Section 2.2.4)

Details of each algorithm will be discussed in the following subsections.

2.2.2 Graph Partitioning

First, we review *graph partitioning* algorithms. Traditionally, the graph clustering problem is related to the graph partitioning problem. In this case, the goal is to partition the graph in such a way to minimize the numbers of edges that cross partitions. This problem is also referred to as the minimum-cut problem and is formally defined as follows:

Problem 2 (Graph partitioning) *Let $\mathcal{P}$ be a partition on graph $\mathbb{G}$, and $ec(\mathcal{P}) = \sum_{u \in \mathbb{V}} ec(u)$ be the size of edge cut, where $ec(u)$ is the number of node u's neighbors that do not belong to the same partition as node u. Graph partitioning divides $\mathbb{V}$ into a set of non-overlapping k partitions $\mathcal{P} = \{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_k\}$ such that (1) $|\mathbb{C}_i| \approx |\mathbb{V}|/k$ for each $\mathbb{C}_i$; (2) $ec(\mathcal{P})$ is minimized.*

Since the graph partitioning problem is NP-hard [84], several approximation algorithms have been derived.

In the simplest case, the 2-way minimum-cut problem, we need to partition the graph into two clusters while minimizing the number of edges that cross the partitions. Ding *et al.* proposed one of the most popular algorithms, called *min-max cut method* [85]. The min-max cut method [85] seeks to partition graph $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$ into two clusters, A and B, in top-down manner. The main idea of min-max cut is minimizing the number of edges between A and B, and maximizing the number of edges within each. A cut is defined as the number of edges that would have to be removed to isolate the nodes in A from those in B. The min-max cut method searches for the partition that creates two clusters whose cut is minimized and while maximizing the number of remaining edges.

A major drawback of the min-max method is that it may not effectively capture the cluster structures included in the given graph. For example, if the min-max method partitions the following two cluster $A = \{u \in \mathbb{V}\}$ and $B = \mathbb{V} \backslash \{u\}$, the two clusters can be an optimum result. Therefore, in practice, the min-max methods requires some constraints such as the size of two clusters A and B should be equal or similar size (*i.e.* $|A| \approx |B|$). However, such constraints are not always appropriate; for instance, in social networks some communities are much larger than others. To address this issue, the *normalized cut method* [86] was proposed by Shi and Malik. This method is an extension of the min-max cut method. In the normalized cut method, Shi and Malik proposed a new measure of partitioning instead of looking the total number of edges connecting two clusters. Specifically,

their measure computes the cut cost as a fraction of the total edge connections to all the nodes in the graph. In order to find such normalized cut partitions, Shi and Malik employed *spectral graph theory* [87]. As a result, the normalized cut method provides partitions, whose size is balanced, as an optimum cluster.

Both the min-max cut method and normalized cut method partition a graph into only two clusters. In order to divide a graph into k clusters, we have to recursively apply these methods in a top-down manner; splitting the graph into two clusters, and then further splitting these clusters, and so on, until k clusters are detected. However, there is no guarantee of the optimality of the recursive clustering results and there is no indicator that is used to halt the bisection procedure.

Kerninghan-Lin algorithm [75], proposed by Kerninghan *et al.* in 1970, is one of the well known techniques for multi-way graph partitioning. Based on hill climbing, this classical algorithm starts with a random cut of the graph. After that, it incrementally swaps nodes among partitions until the number of overall edge-cuts is reduced (this procedure is called *coarsening*). The result of this algorithm may be a local optimum rather than a global optimum. There are many variations based on the Kerninghan-Lin algorithm including the *Fiduccia-Matteyses min-cut heuristic* [88].

Karypis *et al.* proposed *METIS* [72] by improving the Kerninghan-Lin algorithm. METIS works in three steps: (1) coarsening the graph; (2) partitioning the coarsened graph; (3) uncoarsening. In the first step, METIS coarsens a graph by finding the *maximal match*. The maximal match is a set of edges such that all edges in a maximal match do not share their endpoint nodes with the other edges in the maximal match. After it finds a maximal match, it collapses the two ends of each edge into one node, and as a result, the graph is coarsened. The coarsening step is repeated until the graph is small enough. Then, in the second step, it applies the Kerninghan-Lin algorithm or Fiduccia-Matteyses min-cut heuristic directly to the small graph. In the third step, the partitions on the small graph are projected back to the finer graphs. They also proposed a parallel version of METIS named ParMETIS [89].

Recently, Wang *et al.* [90] pointed out that the coarsening approach of traditional partitioning algorithms spoils the semantics of the real-world graphs. The reason is that the correctness of the coarsening approach is based on the following assumption: *A (near) optimal partition on a coarser graph implies a good partitioning of the finer graph*. However, in general, this assumption only holds when the degree of node in the graph is bounded by a constant. As we described in the previous chapter,

real-world graphs tend to follow a power-law degree distribution. Hence, the coarsening may fail to capture the semantics of real-world graphs. In addition, they also pointed out that it is difficult to apply the traditional methods to large-scale graphs with more than billion nodes since the coarsening approach incurs high computation costs. In order to overcome the above limitations, Wang et al. proposed the label propagation based partitioning technique, named *MLP*. MLP uses multilevel label propagation to iteratively coarsen a graph until the coarsened graph is small enough. Since label propagation handles the local topological structure of the given graph, MLP can find "semantic-aware" coarsened graphs. MLP can partition billion-node graphs and is easy to parallelize.

The above approaches have been joined by other proposals such as streaming graph partitioning [91, 92] and approximation algorithms [84, 93].

2.2.3 Modularity-based Clustering

As we described in Section 2.2.2, graph partitioning has drawbacks in that there is no guarantee of the optimality of the clustering result. Furthermore, to permit multi-way partitioning, we have to determine the number of clusters (*i.e.* k in the previous section) that are to be extracted from the graph.

To overcome the above drawbacks, *modularity* was proposed as a measure of graph clustering quality by Newman and Girvan [34]. It measures the difference of a graph structure from an expected random graph. The main idea of modularity is to find groups of nodes that have a lot of inner-group edges and few inter-group edges. Specifically, modularity is defined as follows:

Definition 1 (Modularity Q) *Let e_{uv} be the total number of edges between clusters $\mathbb{C}_u$ and $\mathbb{C}_v$; a_u be the total number of edges that are attached to nodes in cluster $\mathbb{C}_u$; $m = |\mathbb{E}|$. The following equation gives the modularity score of the clustering result.*

$$Q = \sum_{\mathbb{C}_u \in \mathbb{C}} \left\{ \frac{e_{uu}}{2m} - \left(\frac{a_u}{2m} \right)^2 \right\}. \tag{2.1}$$

In Definition 1, $a_u/2m$ is the fraction of edges of cluster $\mathbb{C}_u$ that we would expect to obtain if the graph was a random graph. Therefore, well clustered graphs will have high modularity scores, since the value of e_{uu} is greatly different from that of a random graph.

The main task for modularity-based methods is to find groups of nodes from the graph that maximize modularity Q, formally:

Problem 3 (Modularity clustering) *Let* $\mathbb{C} = \{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_i\}$ *be the set of clusters. Modularity clustering divides the graph into a set of clusters* $\mathbb{C}$ *such that* $\arg\max \sum_{\mathbb{C}_i \in \mathbb{C}} \left\{ \frac{e_{ii}}{2m} - \left(\frac{a_i}{2m}\right)^2 \right\}$.

Although modularity-based algorithms are effective for many applications, finding the maximum modularity involves NP-complete complexity [34]. This problem has led to the introduction of approximate approaches such as top-down algorithm [34], bottom-up greedy algorithms [35–38], simulated annealing approach [94], spectral algorithm [95] and random walk approach [96].

The Girvan-Newman algorithm [34] is a divisive clustering algorithm, which is based on the concept of *betweenness centrality*. Betweenness centrality attempts to identify edges that are critical bridges between different connected components, and delete them, until clusters do not change. The betweenness centrality is defined as the proportion of shortest paths between nodes that pass through a certain edge. Therefore, the betweenness centrality, $\mathcal{B}(e)$, of edge e, is defined as follows:

Definition 2 (Betweenness centrality) *Let* $cp(e, i, j)$ *be the number of shortest paths between node* i *and* j *which pass through edge* e*; and* $sp(i, j)$ *be the number of shortest paths between node* i *and* j*. Betweenness centrality* $\mathcal{B}(e)$ *is defined as follows:*

$$\mathcal{B}(e) = \frac{cp(e, i, j)}{sp(i, j)}. \tag{2.2}$$

The algorithm ranks edges by $\mathbb{B}(e)$, and deletes the edge with the highest score. They used modularity Q as the objective function of the clustering for choosing the number of communities into which a graph should be divided.

Instead of performing an exhaustive search, a bottom-up greedy method named Newman clustering, was proposed by Newman [35]. It iteratively selects and merges pairs of nodes so as to maximize the rise in modularity. It produces reasonable clusters with a hierarchical structure that represents the merge history. Despite its effectiveness in avoiding the NP-complete problem, it incurs high computing cost, $O(|\mathbb{V}|^2)$, where $|\mathbb{V}|$ is the number of nodes.

Various algorithms were proposed to reduce the computational cost of Newman clustering [36–38]. Clauset *et al.* proposed a greedy modularity-based algorithm,

called *CNM* [36], which is one of the most widely used methods recently. They used the modularity gain, which is obtained after merging a pair of nodes, and nested heap structures of modularity gain for all pairs of nodes. It iteratively selects and merges the best pair of nodes, the pair that has the largest modularity gain, from the heap until this procedure no longer improves the modularity. The computation cost of CNM is $O(d|\mathbb{E}| \log |\mathbb{V}|)$, where d and $|\mathbb{E}|$ are the depth of the hierarchical clustering result and the number of edges, respectively. Wakita *et al.* observed that CNM suffers a decrease in clustering speed if the cluster size is unbalanced [37]. Based on this observation, they modified the definition of the modularity gain; specifically, they added the notion of consolidation ratio to the modularity gain. The consolidation ratio measures how the sizes of a pair of clusters differ. By adding this ratio to the modularity gain, they achieved faster clustering than CNM. However, Blondel *et al.* reported that the approaches have a tendency to produce super-clusters with significant low modularity [38]. Since the merge proceeds in a global maximization manner, super-clusters contain a large fraction of the nodes.

Blondel *et al.* proposed the efficient greedy algorithm BGLL [38]. In contrast to CNM, it computes the modularity gain only for adjoining nodes pairs in local maximization. Although BGLL is effective for extracting high modularity clusters, it is difficult to realize quick responses for graphs of unprecedented size, such as Web graphs with their few billion edges. The reason is that it iteratively scans all nodes/edges as long as modularity increases. It is known that the time complexity of BGLL is nearly linear to the edge size.

As described above, modularity based methods can handle significantly large-scale graphs and they are one of the most widely approaches used in many applications [25, 74, 80, 97]. However, despite the efficiency of the modularity-based approach, these methods cannot identify hubs and outliers in graphs. Furthermore, recent research pointed out that modularity is not a scale-invariant measure [40]. Hence, it is difficult for modularity-based methods to find small clusters hidden in large-scale graphs; these methods fail to fully reproduce the ground-truth [41]. This serious problem is well known as the *resolution limit* of modularity [40].

2.2.4 Structural Clustering

Due to the resolution limit of the modularity-based algorithms, *structural clustering* [41, 98–101] has been widely examined in many studies in the last few years. The main idea of structural clustering is that *if adjacent nodes are densely connected to*

each other, they should be assigned to the same cluster. To find clusters based on this idea, these methods extract densely connected subgraphs as clusters. Furthermore, besides extracting clusters, they can find special role nodes, *hubs* and *outliers*, which are not assigned to any cluster. By finding densely connected clusters, hubs, and outliers, they achieve more accurate clustering results than existing algorithms (*e.g.* edge cut based methods and modularity based methods).

In order to evaluate the density of adjacent nodes, Xu *et al.* proposed a measurement called *structural similarity* [102].

Definition 3 (Structural similarity) *The following equation gives the structural similarity, $\sigma(u, v)$, between adjacent nodes u and v.*

$$\sigma(u, v) = \frac{|\mathbb{N}[u] \cap \mathbb{N}[v]|}{\sqrt{|\mathbb{N}[u]||\mathbb{N}[v]|}}, \tag{2.3}$$

where $\mathbb{N}[u] = \{v \in \mathbb{V} : (u, v) \in \mathbb{E}\} \cup \{u\}$.

The structural similarity is a score from 0 to 1 and indicates the scale of matching degree of $\mathbb{N}[u]$ and $\mathbb{N}[v]$. When adjacent nodes share many members of their neighborhoods, their structural similarity will be large.

The main task of the structural clustering is to find all clusters, hubs, and outliers from a graph according to Definition 3 and user-specified parameters.

Yuruk *et al.* proposed *DHSCAN* which adopts the top-down clustering manner [98]. Similar to the Girvan-Newman method in Section 2.2.3, DHSCAN iteratively removes edges based on the ascending order of structural similarity measures given by Definition 3. The graph is divided into disconnected components by removal of edges. This iterative divisive procedure produces a dendrogram showing the hierarchical structure of the clusters. Additionally, a proposal by Feng *et al.* modify Definition 1 to allow this divisive procedure to stop at the point of the maximum structural similarity based modularity [102]. Formal definition of structural similarity based modularity is as follows:

Definition 4 (Structural similarity based modularity Q_s) *Let IS_i be a sum of structural similarity of nodes within cluster $\mathbb{C}_i$; DS_i be the total structural similarity between nodes in cluster $\mathbb{C}_i$ and any nodes in the graph; and TS be the total structural similarity of any adjacent nodes in the graph. The following equation gives the*

structural similarity based modularity score of the clustering result:

$$Q_s = \sum_{\mathbb{C}_i \in \mathbb{C}} \left\{ \frac{IS_i}{TS} - \left(\frac{DS_i}{TS} \right)^2 \right\}. \tag{2.4}$$

The study [98] reports that the clustering results have better quality than those of modularity based algorithms (*i.e* CNM). However, analogous to the Girvan-Newman method, DHSCAN incurs high computation costs for clustering since it has to iteratively compute the structural similarity for all edges. Hence, it is impractical to apply DHSCAN to large-scale graphs.

To achieve high quality clustering results but with lower costs than DHSCAN, some bottom-up algorithms [41, 99, 100, 103, 104] were proposed in the last few years.

SCAN [41], proposed by Xu *et al.*, is the most popular method based on structural similarity. It is an extension of the traditional density based clustering method DBSCAN [105]. This algorithm can find clusters as well as hubs and outliers in a graph once two parameters, ϵ and μ, are specified. For finding clusters, SCAN first extracts seeds of clusters called *cores* from the graph. Cores are formally defined as follows:

Definition 5 (Core) *Let* $\mathbb{N}_\epsilon[u] = \{v \in \mathbb{N}[u] : \sigma(u,v) \geq \epsilon\}$*;* ϵ *and* μ *be user-specified parameters. Node* u *is a core if and only if* $|\mathbb{N}_\epsilon[u]| \geq \mu$*.*

Once SCAN determines node u to be a core, it assigns all nodes in $\mathbb{N}_\epsilon[u]$ to the same cluster as the core. It then selects a node from the cluster and repeatedly finds cores and expands clusters from the cores. After the cluster extraction terminates, SCAN classifies the remaining nodes as hubs or outliers. If a remaining node bridges several clusters, it is regarded as a hub; otherwise it is an outlier. This algorithm incurs the average cost of $O(|\mathbb{E}|^2/|\mathbb{V}|)$. The reason is that SCAN evaluates the structural similarities for all adjacent nodes, and each similarity computation requires $O(|\mathbb{E}|/|\mathbb{V}|)$ time, on average.

Huang *et al.* and Sun *et al.* proposed the parameter-free methods named SHRINK [99] and gSkeletonClu [100], respectively. SHRINK is a parameter-free hierarchical clustering algorithm that combines the advantages of structural similarity and modularity based methods. It first computes structural similarities for all adjacent nodes. Then it aggregates densely connected node pairs, called *micro communities*, into the same clusters if the aggregation improves the Q_s score given by Definition

4. In this way, SHRINK achieves parameter-free and hierarchical clustering. In contrast, gSkeletonClu tries to find the best value of ϵ in the clustering process. Following SHRINK, it first computes the structural similarities for all adjacent nodes in the graph. After that, it extracts spanning trees [106] from the graph by using the results of the structural similarities; and then searches the trees to find the effective values of ϵ. SHRINK and gSkeletonClu are user-friendly algorithms since they do not require user-specified parameters. Moreover, Huang *et al.* reported that their clustering quality is better than that of SCAN [99–101]. However, these methods require exhaustive similarity computations for all adjacent nodes; hence the time complexities of SHRINK and gSkeletonClu are at least $O((|\mathbb{E}|^2 \log |\mathbb{V}|)/|\mathbb{V}|)$ and $O(|\mathbb{E}|^2/|\mathbb{V}| + |\mathbb{V}| \log |\mathbb{V}|)$, respectively. Therefore, as in SCAN, both SHRINK and gSkeletonClu require large computation time for large-scale graphs.

Recently, Lim *et al.* proposed LinkSCAN* [104], which uses SCAN to find overlapping communities from a graph. In order to detect overlapping communities very accurately, LinkSCAN* transforms the graph into a link-space graph, which combines the advantages of the graph and line graph [107]. However, this transformation entails an increase of the size of the graph for clustering. Hence, they introduced a *graph sampling* step. This approach is certainly efficient in reducing the computation time; to the best of our knowledge, LinkSCAN* is one of the most efficient methods for structural similarity based clustering. However, it degrades the clustering results compared to SCAN since sampling involves approximation.

2.2.5 Summary of the Survey

This chapter provided an overview of the key techniques used for cluster analysis of large-scale graphs. In this chapter, we investigated existing algorithms that are grouped into three types; graph partitioning, modularity-based clustering, and structural clustering. Most of the initial graph clustering algorithms tried to minimize the number of edges that cross different partitions. In order to capture more complicated cluster and community structures hidden in graphs, modularity-based clustering and structural clustering were widely developed in the fields of computer science and physics.

Our brief survey indicates that existing methods still suffer from their computational cost at finding clusters in billion-node graphs. The reason is that the methods require exhaustive computations; they have to evaluate at least all edges in graphs. In order to avoid the exhaustive computations of traditional algorithms, this the-

Table 2.1: Summary of graph clustering algorithms

	Algorithm	Complexity
modularity clustering	Girvan and Newman [34]	$O(\|\mathbb{E}\|^2\|\mathbb{V}\|)$
	Newman [35]	$O(\|\mathbb{V}\|^2)$
	Clauset *et al.* [36]	$O(d\|\mathbb{E}\| \log \|\mathbb{V}\|)$
	Blondel *et al.* [38]	$O(\tau\omega\|\mathbb{E}\|)$
	IMAC (Chapter 3)	$O(\|\mathbb{E}\| + \frac{\alpha(1-c)}{\alpha-1}\|\mathbb{V}\|)$
	ParBGLL (Chapter 4)	$O(\frac{1}{p}\|\mathbb{E}\| + \|\mathbb{V}\|)$
structural clustering	Xu *et al.* [41]	$O(\|\mathbb{E}\|^2/\|\mathbb{V}\|)$
	Huang *et al.* [99]	$O((\|\mathbb{E}\|^2 \log \|\mathbb{V}\|)/\|\|\mathbb{V}\|)$
	Sun *et al.* [100]	$O(\|\mathbb{E}\|^2/\|\mathbb{V}\| + \|\mathbb{V}\| \log \|\mathbb{V}\|)$
	LinkSCAN* [104]	$O(\|\mathbb{E}\|)$
	SCAN++ (Chapter 5)	$O(\frac{1-c}{c}\|\mathbb{E}\|)$

sis presents three efficient algorithms for modularity-based and structural clustering. Differ from the existing studies, our proposals are designed to avoid exhaustive computations by capturing the structural properties described in Chapter 1. In Chapter 3, we present an efficient modularity-based clustering algorithm, named *IMAC*, which utilizes clustering coefficient and power-law degree distribution for avoid exhaustive modularity computations. Chapter 4 also presents a fast vectorized modularity-based graph clustering approach *ParBGLL*. ParBGLL reduces the computation cost such as the number of CPU instructions for each modularity computations by using data parallel scheme of modern CPUs. Differ from Chapter 3 and 4, we present an efficient algorithm *SCAN++* for structural clustering in Chapter 5. SCAN++ effectively reduces number of structural similarity computations and computational costs of each similarity computation by capturing the clustering coefficient of graphs.

Table 2.1 lists the time complexities of major modularity-based clustering and structural clustering methods including our proposals. Note that IMAC, ParBGLL and SCAN++ are the methods proposed in Chapter 3, 4 and 5, respectively. The symbols $|\mathbb{V}|$ and $|\mathbb{E}|$ are the number of nodes and edges in the graph, respectively. Also, d, c and n are dendrogram depth, clustering coefficient, and average degree, respectively. The symbols τ and ω in BGLL are the number of iterations consumed in its modularity computations and node aggregations, respectively. We omitted

the time complexity of DHSCAN and the approach proposed by Wakita and Tsumi since no full discussion has been provided. As shown in Table 2.1, our proposals achieve lower time complexity than the traditional algorithms since they using the structural properties for avoiding exhaustive computations. The details of each proposal are discusses in the following chapters.

Chapter 3

Fast Algorithm for Modularity-based Graph Clustering

In AI and Web communities, modularity-based graph clustering algorithms are being applied to various applications. However, existing algorithms are not applied to large graphs because they have to scan all nodes/edges iteratively. The goal of this chapter is to efficiently compute clusters with high modularity from extremely large graphs with more than a few billion edges. The heart of our solution is to compute clusters by incrementally pruning unnecessary nodes/edges and optimizing the order of vertex selections. Our experiments show that our proposal outperforms all other modularity-based algorithms in terms of computation time, and it finds clusters with high modularity.

3.1 Introduction

Recently, modularity-base clustering proposed by Newman and Girvan [34] has become one of the most popular algorithms for extracting clusters in a graph. Modularity evaluates the density of edges inside clusters as compared to edges between clusters. The better clustering results are achieved with higher modularity scores. Modularity-based algorithms have been applied to many applications such as image segmentation [97] and brain analysis [25] due to its effectiveness.

Although the modularity-based algorithms are effective for many applications,

finding the maximum modularity involves NP-complete complexity [34]. This problem has led to the introduction of approximation approaches [35–38]. Blondel *et al.* proposed an efficient greedy algorithm BGLL [38]. To the best of our knowledge, BGLL is representative for the state-of-the-art algorithm; it achieves fast clustering with higher modularity than the other algorithms. In contrast to CNM, it computes the modularity gain only for the adjoined nodes pairs as a local maximization. Blondel *et al.* reported that BGLL requires almost 3 hours to process graphs with 118 million nodes [38]. Although BGLL is effective for extracting high modularity clusters, it is difficult for BGLL to realize quick responses for graphs of unprecedented size, such as Web graphs with their few billion edges. This is because it iteratively scans all nodes/edges as long as the modularity is increasing.

To overcome the limitation of computing time in the previous approaches, we propose a novel clustering algorithm. In order to reduce computational cost, we introduce three ideas. First, we incrementally aggregate nodes, which are placed in a same cluster, into a single vertex. Second, we incrementally prune computations of modularity gain for nodes whose clusters are obviously obtained. Last, we optimize the order of vertex selections for efficient clustering. Our proposal has the following attractive characteristics:

- **Efficiency**: The proposed algorithm is considerably faster than existing approaches such as CNM and BGLL.
- **High-modularity**: Our approach provides clustering results with high modularity; it returns almost the same modularity scores as the state of the art approach, BGLL.
- **Effectiveness**: Our algorithm is effective in improving the performance for large-scale complex networks.

To the best of our knowledge, our approach is the first solution to divide graphs into clusters that have more than 100 million nodes and 1 billion edges within 3 minutes. These characteristics confirm the practicality of our algorithm for real world applications. With our proposal, many more applications can be implemented more efficiently.

Table 3.1: Definition of main symbols

Symbol	Definition
$\Gamma(u)$	Set of adjacent nodes of node u
$\mathbb{P}_i$	Set of prunable nodes defined by Definition 10
$\mathbb{T}_i$	Set of target nodes defined by Definition 11
Q	Modularity score of a clustering result given by Definition 6
$\triangle Q_{ij}$	Modularity gain between cluster $\mathbb{C}_i$ and $\mathbb{C}_j$ defined by Definition 7
e_{ij}	Number of edges that bridges cluster $\mathbb{C}_i$ and $\mathbb{C}_j$
a_i	Number of edges that connected to nodes in cluster $\mathbb{C}_i$. $a_i = \sum_k e_{ik}$
c_u	Cluster ID of the cluster to which node u belongs
c	score of clustering coefficient
α	skewness of the power-law degree distribution

3.2 Preliminary

In this section, we give some preliminaries of this chapter. Table 3.1 lists main symbols and their definitions that are used in this chapter.

3.2.1 Modularity

Modularity, introduced by Newman *et al.* [34], is widely used to evaluate the cluster structure of a graph from a global perspective. It measures the differences in graph structures from an expected random graph. The main idea of modularity is to find groups of nodes that have a lot of inner-group edges and few inter-group edges. Modularity Q is formally defined as follows:

Definition 6 (Modularity Q) *Let e_{uv} be the total number of edges between cluster $\mathbb{C}_u$ and $\mathbb{C}_v$; a_u be the total number of edges that are attached to nodes in cluster $\mathbb{C}_u$; and m be the total number of edges in the whole graph. The following equation gives the modularity score of the clustering result.*

$$Q = \sum_{\mathbb{C}_u \in \mathbb{C}} \left\{ \frac{e_{uu}}{2m} - \left(\frac{a_u}{2m} \right)^2 \right\}. \tag{3.1}$$

In Definition 6, $a_u/2m$ is the expected fraction of edges of $\mathbb{C}_u$, which can be obtained when we assume the graph to be a random graph. Therefore, well clustered

graphs will have high modularity scores, since the value of e_{uu} is highly different from the random graph.

3.2.2 BGLL

We then overview the state-of-the-art algorithm BGLL.

BGLL is divided in two phases that are repeated iteratively. First phase is local clustering phase. In this phase, BGLL finds clusters by greedily maximizing the modularity of its clustering result. Specifically, it first picks a node and computes modularity gains, which is an increment of modularity after assigning two nodes into a same cluster, for all adjacent nodes of the node. If BGLL finds adjacent nodes that have the largest positive score of modularity gain among adjacent nodes, it assigns the node into the same cluster of the adjacent node. Otherwise, the node stays in its original cluster. BGLL continues this local clustering until there are no improvements of modularity.

Second phase is cluster aggregation phase. In the cluster aggregation phase, BGLL aggregates each cluster, which is obtained by the local phase, into a node. Besides the cluster aggregation, it also merges edges that are lying inner and inter clusters into weighted edges. The inner edges of a cluster are merged into a weighted self-loop edge of the corresponding aggregated node. The weight of the self-loop edge equals to doubled number of edges that are included in the cluster. On the other hand, the inter edges between two clusters are merged into a weighted edge that bridges the corresponding aggregated nodes. The weight of the weighted edge is obtained from number of edges that bridge two clusters. After finishing cluster aggregation phase, we obtain a weighted graph.

BGLL iterates the above two phases until the modularity score of its clustering results does not increase. Algorithm 1 shows the detail procedure of BGLL.

3.3 Proposed method

This section presents details of our proposal. In contrast to all the other algorithms, we can find clusters with high modularity in graphs of unprecedented size, such as more than a few billion of edges, within a few minutes. We give an overview the

Algorithm 1 BGLL

Input: $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$;
Output: clustering result of $\mathbb{G}$;

repeat
  repeat
    select node u from $\mathbb{V}$;
    for all node v included in the adjacent nodes of node u **do**
      compute modularity gain between node u and v;
    end for
    find node v' that has the largest modularity gain from the adjacent nodes of node u;
    if the modularity gain of node $v' > 0$ **then**
      assign node u to the cluster of node v';
    else
      stay node u in its original cluster;
    end if
  until modularity Q is increased
  aggregate clusters into a weighted graph;
until modularity Q is increased

ideas underlying our algorithm that is followed by a full description including graph clustering algorithm.

3.3.1 Ideas

We introduce three ideas to avoid the high computation cost of existing algorithms. First, we incrementally aggregate nodes, which are placed in a same cluster, into a single node to eliminate unnecessary nodes/edges from the graph. Second, we incrementally prune nodes whose clusters are obtained without modularity computing. Last, we optimize the order of node selections to reduce the number of modularity computations in the clustering process. Instead of iterative computations for all nodes/edges in the whole graph, we only compute the key nodes/edges efficiently. Moreover, our proposal successfully produces clustering results with high modularity by obtaining clusters in a local modularity maximization and avoiding skewed access.

These simple ideas have two main advantages. First, we can extract clusters with quite-small computational cost for complex networks [108]. Our ideas successfully handle the interesting characteristics of complex networks; high clustering coefficients and power-law degree distributions. This is because our ideas are designed to perform well even if the graph has many co-occurrence nodes between adjacent nodes. That is, high clustering coefficients lead our algorithm to compute efficiently. Additionally, our ideas perform well in the case that there are strong imbalances among the degrees of all nodes as in power-law distributions. Thus,

our algorithm runs faster on large size complex networks than the state of the art algorithm.

Second, our algorithm can produce clustering results with high modularity by not missing the chances that may improve the modularity. The reason is twofold. First is that our pruning method does not sacrifice modularity. Second is that our ideas successfully prevent our algorithm from producing imbalanced clustering results, which would otherwise greatly degrade the modularity. Therefore, our algorithm can extract clustering results with high modularity.

3.3.2 Incremental aggregation

We extract clusters by incrementally aggregating nodes placed in a same cluster into an equivalent single node with weighted edges. In contrast to previous algorithms, our proposal does not traverse all nodes/edges multiple times. In this section, we formally introduce our incremental aggregation technique and its properties.

We specify the modularity gain proposed by Newman [35], since it is also utilized by our algorithm.

Definition 7 (Modularity gain $\triangle Q_{uv}$) *Let $\triangle Q_{uv}$ be the modularity gain which is obtained after merging nodes u and v. The modularity gain $\triangle Q_{uv}$ is defined as follows:*

$$\triangle Q_{uv} = 2\left\{\frac{e_{uv}}{2m} - \left(\frac{a_u}{2m}\right)\left(\frac{a_v}{2m}\right)\right\}. \tag{3.2}$$

By using modularity gain, our proposal finds clusters in a local maximization manner. When node u is selected, it computes the modularity gain of u for each v in $\Gamma(u)$, where $\Gamma(u)$ is the set of nodes neighboring u. After computing all modularity gains between u and v, our algorithm incrementally aggregates u and v that yields the highest rise in modularity. Details of the incremental aggregation and aggregated node are as follows:

Definition 8 (Incremental aggregation) *Let node v be the neighboring node of node u that yields the highest rise in modularity. If node u has a positive modularity gain for v (i.e. $\triangle Q_{uv} > 0$), the pair of nodes, u and v, are aggregated into a single node w. If node v has the negative modularity gain (i.e. $\triangle Q_{uv} \leq 0$), the pair of nodes u and v are not aggregated.*

Definition 9 (Aggregated node) *We initialize the weight of each edge to 1 in the given graph. If node w is aggregated from node u and v, node w has two types of weighted edges; a self-loop edge and outer edges. The weight of the self-loop edge is obtained by summing (1) weights of the self-loop edges of nodes u and v, and (2) weights of edges between node u and v. The weights of outer edges for other nodes are obtained by summing the weights of edges that incident nodes u and v.*

From Definition 9, the degree of w is given as the number of the weighted outer edges that are obtained by aggregating nodes/edges included in the same cluster. Then, we introduce the theoretical properties of Definition 8 and 9.

Lemma 1 (Equivalence of the modularity) *If (1) node u and v belong to the same cluster (i.e. $c_u = c_v$) and (2) node u and v are aggregated into node w, the modularity taken from node w is equivalent that of nodes u and v.*

Proof Let $Q_{(u,v)}$ be the modularity for the case that node u and v belong to the same cluster, and Q_w be the modularity of aggregated node w. From Definition 9, the weighted edges of w are $e_{ww} = e_{uu} + e_{vv} + 2e_{uv} = e_{(u,v)(u,v)}$ and $a_w = a_u + a_v = a_{(u,v)}$. Q_w is obtained as follows:

$$Q_w = \frac{e_{ww}}{2m} - \frac{a_w^2}{4m^2} = \frac{e_{uu} + e_{vv} + 2e_{uv}}{2m} - \frac{(a_u + a_v)^2}{4m^2} = Q_{(u,v)} \tag{3.3}$$

Thus, node w has the same modularity as nodes u and v that belong to the same cluster. □

From Lemma 1, we can reduce the number of nodes/edges in the graph without sacrificing modularity quality. Additionally, we advance the following lemma to avoid iterative traversal of all nodes/edges:

Lemma 2 (Negativity of the modularity) *Once a node has negative modularity gain for all neighbors, it will never be clustered with its neighbors in the subsequent process.*

Proof We assume nodes v_i and v_j are connected. There are two cases in which the modularity gains of node u can be updated. First is that node u is connected to both of v_i and v_j. In this case, the modularity gains of u are given as $\triangle Q_{uv_i} < 0$ and $\triangle Q_{uv_j} < 0$, respectively. If v_i and v_j are aggregated into node w, the updated

modularity gain $\triangle Q_{uw}$ can be obtained by Definition 7 and 9 as follows:

$$\begin{aligned}
\triangle Q_{uw} &= 2\left\{\frac{e_{uw}}{2m} - \left(\frac{a_u}{2m}\right)\left(\frac{a_w}{2m}\right)\right\} && (3.4)\\
&= 2\left\{\frac{e_{uv_i} + e_{uv_j}}{2m} - \left(\frac{a_u}{2m}\right)\left(\frac{a_{v_i} + a_{v_j}}{2m}\right)\right\} && (3.5)\\
&= 2\left(\triangle Q_{uv_i} + \triangle Q_{uv_j}\right) < 0. && (3.6)
\end{aligned}$$

As can be see, node u always has negative modularity gain after merging pairs of neighbor nodes in this case. Next, we assume the case that only node v_i is connected to node u. In this case, the modularity gain between u and v_i is given as $\triangle Q_{uv_i} < 0$. If v_i and v_j are aggregated into a node w, the updated modularity gain $\triangle Q_{uw}$ is obtained as follows:

$$\begin{aligned}
\triangle Q_{uw} &= 2\left\{\frac{e_{uw}}{2m} - \left(\frac{a_u}{2m}\right)\left(\frac{a_w}{2m}\right)\right\} && (3.7)\\
&= 2\left\{\frac{e_{uv_i}}{2m} - \left(\frac{a_u}{2m}\right)\left(\frac{a_{v_i} + a_{v_j}}{2m}\right)\right\} && (3.8)\\
&= 2\left(\triangle Q_{uv_i} - \frac{a_u a_{v_j}}{4m^2}\right) < 0. && (3.9)
\end{aligned}$$

This case also has no positive improvement of $\triangle Q_{uw}$ after aggregation. Therefore, node u never finds a neighbor node yielding positive modularity gain. Thus, once a node only has negative modularity gains, it will never be clustered in the subsequent process. □

From Lemma 2, we can efficiently reduce the number of traverses for all nodes/edges. This is because, once a node is detected as yielding having only negative modularity gain, it is never considered for aggregation thereafter.

Our proposal efficiently handles the structural feature of high clustering coefficient. The clustering coefficient of adjoined nodes [109] measures how close the pair of nodes is to being a clique. By considering this pairwise clustering coefficient, the efficiency of our algorithm is confirmed as follows:

Lemma 3 (Efficiency of incremental aggregation) *Let c be the clustering coefficient of the adjacent nodes, and n be the number of neighbors adjoined to the pair of nodes (i.e. $|\Gamma(u) \cup \Gamma(v)|$). In each incremental aggregation, our algorithm can eliminate cn edges.*

Proof If a pair of nodes have n neighboring nodes, the pair is expected to have cn neighboring nodes that are co-referenced from both of the pair. From Definition 9, cn edges, indicates co-referenced nodes from the pair, will be eliminated by

aggregating the pair into a single node. Therefore, we can eliminate cn edges in each aggregation. □
From Lemma 3, it is obvious that our algorithm performs well when the given graph has a high clustering coefficient.

3.3.3 Incremental pruning

In practice, there are a lot of nodes whose clusters are trivially determined, we call them *prunable nodes*. We call the set of nodes whose modularity gains are to be computed as *target nodes*, in the clustering process. Unlike existing algorithms, our algorithm computes the modularity gain for only target nodes by dynamically removing prunable nodes in incremental manner. We formally introduce below the definitions of prunable nodes and target nodes with their theoretical properties. The set of prunable nodes $\mathbb{P}_i$ in the i–th aggregation is defined as follows:

Definition 10 (Prunable vertices) *Let c_u be a cluster to which node u belongs. The following equation gives the set of prunable nodes in the i–th aggregation.*

$$\mathbb{P}_i = \begin{cases} \emptyset & (i = 0) \\ \{u : |\Gamma(u)| = 1\} \cup \{u : \forall v, w \in \Gamma(u), c_v = c_w\} & (i > 0) \end{cases} \tag{3.10}$$

Definition 10 indicates that a node is included in $\mathbb{P}_i$ if (1) the node has only a single adjacent node, or (2) all the adjacent nodes of the node belong to the same cluster. We can introduce the following properties of prunable nodes:

Lemma 4 (Non-negativity of the modularity gain) *If node u is included in $\mathbb{P}_i$, the modularity gain of node u for each adjacent node must be greater than 0.*

Proof From Definition 10, if the node included in $\mathbb{P}_i$ has only a single adjacent node in the given graph (*i.e.* $|\Gamma(u)| = 1$), we have $e_{uv} = a_u = 1$ and $0 < a_v < 2m$. Therefore, from Definition 7, the modularity gain $\triangle Q_{uv}$ between node u and v is always greater than 0. If all neighbor nodes of node u belong to the same cluster c_w, (*i.e.* $c_v = c_w, \forall v, w \in \Gamma(u)$), node u has edges $e_{uw} = a_u > 0$ and cluster c_w has $0 < a_w < 2m$. Therefore, the modularity gain $\triangle Q_{uw}$ between node u and cluster c_w is always greater than 0. □
From Lemma 4, all nodes included in $\mathbb{P}_i$ must have positive modularity gains. That is, all nodes that belong to $\mathbb{P}_i$ are always clustered in their neighbors' cluster. Therefore, we can aggregate these nodes without sacrificing the quality of modularity.

From Lemma 4, the set of target nodes in the i–th aggregation, $\mathbb{T}_i$, can be defined as follows:

Definition 11 (Target vertices) *The following equation gives the set of target nodes in the i-th aggregation.*

$$\mathbb{T}_i = \begin{cases} \mathbb{V} & (i = 0) \\ \mathbb{T}_{i-1} - \mathbb{P}_i & (i > 0) \end{cases} \tag{3.11}$$

Our algorithm incrementally prunes $\mathbb{P}_i$ from $\mathbb{T}_{i-1}$ in each aggregation step. However, computation costs would be excessive if we naively search for nodes in $\mathbb{P}_i$ such that all of their adjacent nodes belong to a same cluster. For efficient computing, therefore, we introduce a theoretical property of $\mathbb{T}_i$ and $\mathbb{P}_i$ as follows:

Lemma 5 (Incremental pruning) *We can find all nodes included in $\mathbb{P}_i$ by obtaining nodes such that they have only a single adjacent node from $\mathbb{T}_{i-1}$ in each aggregation.*

Proof If a node in the given graph has only a single adjacent node, the node can be obviously pruned. If all neighboring nodes of a node belong to a same cluster, all of them have already been aggregated into a single node by Definition 8. Therefore, we can obtain $\mathbb{P}_i$ by finding nodes such that they have only a single adjacent node.□ By Lemma 5, we can efficiently find all nodes in $\mathbb{P}_i$.

3.3.4 Efficient ordering of node selections

We establish efficient ordering of node selections for local modularity maximization to reduce the computations. Our proposal dynamically selects node with the smallest degree by handling the power-law degree distribution.

One of the famous properties of complex graphs is the power-law degree distribution [110]; most nodes have relatively few neighbors while a few nodes have many neighbors. Under the power-law degree distribution, the frequency of nodes with degree number of d is proportional to $d^{-\alpha}$, where exponent α is a positive constant that represents the skewness of the degree distribution. A high α implies that the vast majority of nodes have small degree. As α decreases, the graph density and the number of large degree nodes increases. Given the power-law degree distribution, we find the following empirical observation:

- **Observation 1**: Greedy modularity-based algorithms, which maximize modularity in a local manner, can extract clusters with a small number of modularity computations by selecting nodes that have the smallest degree.

We compute the modularity gains of node u for all neighbor nodes in $\Gamma(u)$. This process involves $|\Gamma(u)|$ times modularity computations for node u to find the node that yields the highest rise in modularity gain. Therefore, selecting nodes that have a large degree waste computation time. Thus, we find nodes of the highest modularity gain with low computation cost by dynamically selecting nodes with the smallest degree. By combining the ordering and the incremental aggregation, we reduce the size of degrees. Thus, the ordering reduces the computational cost especially for high degree node. Additionally, we find the node of the highest modularity gain more efficiently as the graph strongly follows power-law degree distribution. This is because nodes in the power-law degree distribution tend to have highly skewed degree.

Moreover, we obtain clusters with high modularity by avoiding skewed access to nodes of large degree. This is because our proposal successfully prevents the results from producing super-clusters, which would otherwise greatly degrade the modularity. Thus, we extract clusters from graphs with high modularity.

3.3.5 Graph clustering algorithm

Algorithm 2 shows our algorithm. First, if $i = 0$, the algorithm initializes $\mathbb{P}_0 = \emptyset$ and $\mathbb{T}_0 = \mathbb{V}$ based on Definition 10 and 11, respectively (line 1). Next, it incrementally computes prunable nodes $\mathbb{P}_i$ (line 4) and merge each node in $\mathbb{P}_i$ into its neighboring cluster (lines 5-7). Next, it obtains target nodes $\mathbb{T}_i$ as described in Lemma 5 and Definition 11 (line 8). It selects node u with the smallest degree from $\mathbb{T}_i$ based on Observation 1 (line 9), and finds neighbor node v that maximizes the modularity gain as defined in Definition 7 (line 10). If the modularity gain $\triangle Q_{uv}$ is positive, it then aggregates nodes u and v into a single node as described in Definition 8 and 9 (lines 11-14). Otherwise, node u is pruned from $\mathbb{T}_i$ by Lemma 2 (line 16). If $\mathbb{T}_i$ contains no nodes, it terminates its iteration cycle. Finally, it returns aggregated nodes as a result; all nodes included in an aggregated node belong to same cluster.

We provide a theoretical analysis of the computation cost.

Algorithm 2 Incremental Modularity Agglomerative Clustering (IMAC)

Input: $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$;
Output: clustering result of $\mathbb{G}$;
$i = 0, \mathbb{P}_0 = \emptyset, \mathbb{T}_0 = \mathbb{V}$;
while $|\mathbb{T}_i| > 0$ **do**
  $i = i + 1$;
  $\mathbb{P}_i = \{u : |\Gamma(u)| = 1\}$;
  for $\forall u \in \mathbb{P}_i$ **do**
    aggregate node u into its neighbor;
  end for
  $\mathbb{T}_i = \mathbb{T}_{i-1} - \mathbb{P}_i$;
  select node u from $\mathbb{T}_i$ that has the smallest degree;
  $v = \arg\max_{v'} \triangle Q_{uv'}$;
  if $\triangle Q_{uv} > 0$ **then**
    aggregate u and v into a single node w;
    $\mathbb{T}_i = \mathbb{T}_i - \{u, v\}$;
    $\mathbb{T}_i = \mathbb{T}_i \cup \{w\}$;
  else
    $\mathbb{T}_i = \mathbb{T}_i - \{u\}$;
  end if
end while

Theorem 1 (Computational cost) *Our algorithm requires* $O(|\mathbb{E}| + \frac{\alpha(1-c)}{\alpha-1}|\mathbb{V}|)$ *time to obtain a clustering result from a graph, where* c *and* α *are the clustering coefficient and the skewness of the degree distribution, respectively.*

Proof Our algorithm needs $2|\mathbb{E}|$ computations without the incremental aggregation, because it has to compute $\triangle Q$ for all neighbors for each node. Lemma 3 shows that each incremental aggregation eliminates cn edges from the given graph, where n is the average degree. Moreover, it iterates the incremental aggregations $|\mathbb{T}_i| \approx |\mathbb{V}|$ times by Definition 8 and 9, so we can eliminate $cn|\mathbb{V}|$ edges from the given graph. As a result, our method requires $O(|\mathbb{E}| - cn|\mathbb{V}|)$ modularity computations for clustering the given graph. In addition to the modularity computation costs, our method also requires aggregation costs for merging two nodes/clusters. The computational cost of each aggregation is $O(n|\mathbb{V}|)$ since our algorithm traverses all adjacent nodes for merging two nodes. Hence, the total computational cost is $O(|\mathbb{E}| + n(1-c)|\mathbb{V}|)$. Generally, the average degree is $n = \frac{\alpha}{\alpha-1}$ for the real-world graphs that follow the power-law degree distribution. Therefore, the computational cost of our method is $O(|\mathbb{E}| + \frac{\alpha(1-c)}{\alpha-1}|\mathbb{V}|)$. □

This theorem indicates that the computation cost of our proposal is dramatically smaller than those of existing algorithms; for instance, CNM requires $O(d|\mathbb{E}| \log |\mathbb{V}|)$ to obtain a clustering result. Furthermore, we have even smaller computation cost than the one described in Theorem 1 in practical cases. This is because we have two other techniques to enhance the clustering speed; incremental pruning and efficient ordering. However their computation costs strongly depend on the structure of

the graph, we show concrete computation times for real world datasets in the next section.

3.4 Experimental evaluation

We conducted evaluations to confirm the effectiveness of our algorithm. In the experiments, we used the five public datasets [111] to evaluate our algorithm:

- *dblp-2010*: This scientific collaboration graph was extracted from the bibliography service DBLP in 2010; each node is a scientist and each edge is coauthor relationship.
- *ljournal-2008*: This graph was obtained from a social networking site LiveJournal in 2008; each node and edge represent a user and friendship among users, respectively.
- *uk-2005*: This graph was obtained from a 2005 crawl of .uk domain; each node and edge mean a Web page and a link between pages, respectively.
- *webbase-2001*: This Web graph of .us domain in 2001 was downloaded from the Stanford Webbase project, each node and edge mean a Web page and a link, respectively.
- *uk-2007-05*: This graph is a expansion of *uk-2005* crawled from .uk domain Web pages in May, 2007.

The details of our datasets are shown in Table 5.2, where α is the exponent that controls the skewness of the degree distribution, described in previous section. Additionally, we also use synthetic datasets generated by DIGG[1] and LFR benchmark [112] to evaluate the effectiveness of our proposal for complex networks. The details setting will be described later.

All experiments were conducted on a Linux 2.6.18 server with Intel Xeon CPU L5640 2.27GHz and 144GB RAM. We implemented our proposal using C++ and we used the optimization parameter "-O2" for each algorithm. To evaluate the existing algorithms, we used programs of CNM[2] and BGLL[3] published on their authors'

[1] http://digg.cs.tufts.edu/

[2] http://www.cs.unm.edu/ aaron/research/fastmodularity.htm

[3] https://sites.google.com/site/findcommunities/

Table 3.2: Real-world datasets

	dblp-2010	ljournal-2008	uk-2005	webbase-2001	uk-2007-05
$\|\mathbb{V}\|$	326,186	5,363,260	39,459,925	118,142,155	105,896,555
$\|\mathbb{E}\|$	1,615,400	79,023,142	936,364,282	1,019,903,190	3,738,733,648
α	2.82	2.29	1.71	2.14	1.51

Table 3.3: Modularity Q for each dataset

	dblp-2010	ljournal-2008	uk-2005	webbase-2001	uk-2007-05
Proposed	0.90	0.74	0.98	0.98	0.97
BGLL	0.88	0.74	0.97	0.96	0.97
CNM	0.82	-	-	-	-

sites.

3.4.1 Efficiency

We evaluated the clustering performance of each algorithm through wall clock time for each real world dataset. Figure 3.1 shows the results of computational time. In Figure 3.1, our proposal is tested under two different types; *Proposed* and *Proposed-Limited*. *Proposed* represents the full version of our algorithm described in Algorithm 2. And *Proposed-Limited* represents the limited version of Algorithm 2 which only uses the incremental aggregation. Since CNM cannot compute clusters in a day except for *dblp*, we omitted the results of CNM for other dataset. Figure 3.1 indicates that *Proposed* is significantly faster than the other algorithms under all conditions examined. As described earlier, the existing algorithms traverse all nodes/edges multiple times while our algorithm dynamically eliminates nodes/edges. As a result, our proposal is up to 60 times faster than the state of the art algorithm BGLL; our algorithm computed clusters from the graph with 1 billion edges in 156 seconds. Furthermore, *Proposed-Limited* is up to 20 times faster than BGLL, even though *Proposed* is almost three times faster than *Proposed-Limited* under all conditions. This indicates that the incremental aggregation contributes most to the improvement. *Proposed* more efficiently reduces the computational cost than *Proposed-Limited* by combining incremental aggregation, incremental pruning and efficient ordering.

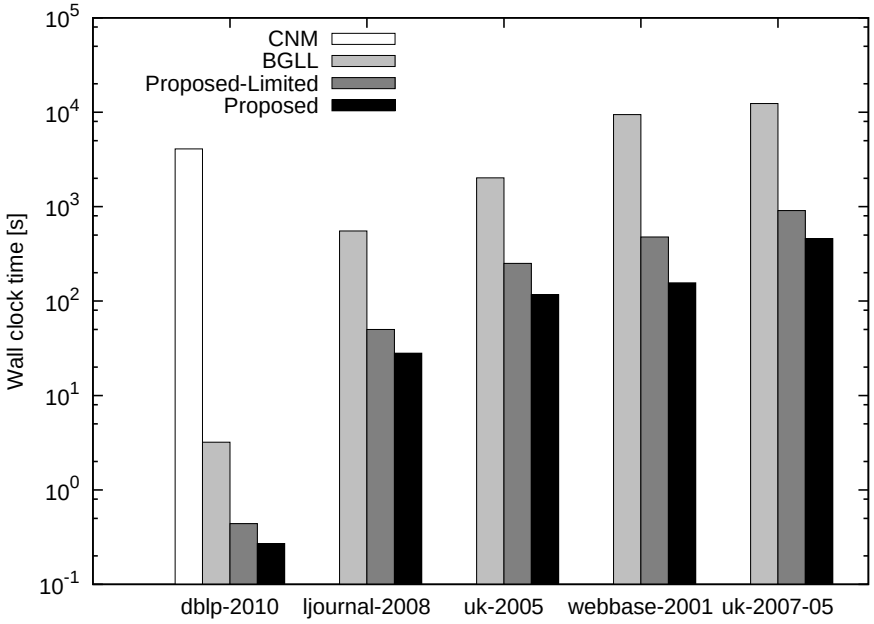

Figure 3.1: Clustering time for real-world datasets

3.4.2 Modularity

One major advantage of our algorithm is that it outputs clusters with high modularity. Table 3.3 shows modularity Q for each of the real world datasets. Table 3.3 indicates that the modularity score of our algorithm is higher than that of CNM. Since CNM optimizes modularity in a global manner, it tends to produce super-clusters which significantly degrades modularity. In contrast, our algorithm successfully avoid to produce super-clusters by using a local modularity maximization and efficient ordering of node selections. Furthermore, Table 3.3 shows that our proposal achieves slightly higher modularity than BGLL even though BGLL also performs higher modularity than CNM. The computation time of BGLL is significantly larger than ours as shown in Figure 3.1. That is, these results show the superiority of our approach over the previous approaches.

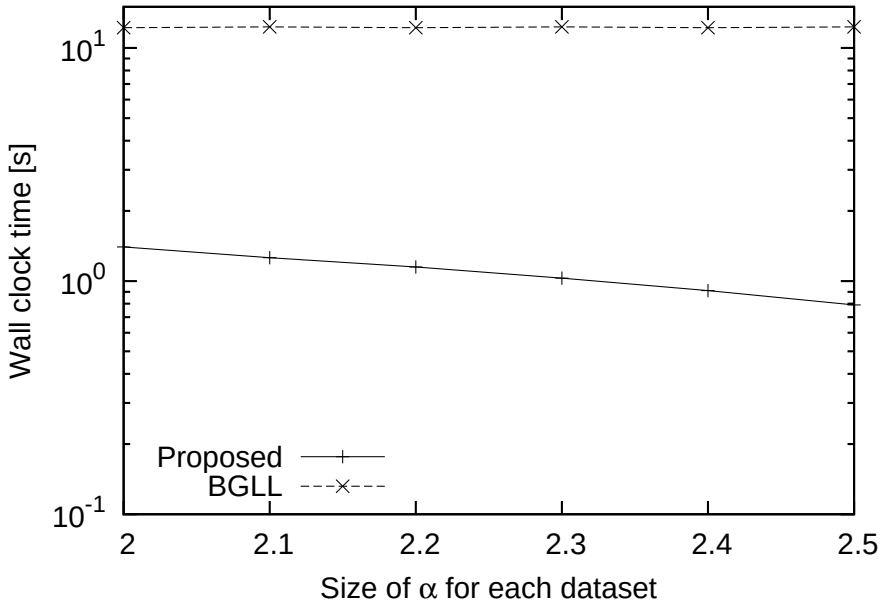

Figure 3.2: Power-law differences

3.4.3 Effectiveness

We evaluate the effectiveness of our algorithm for complex networks that have high cluster coefficients and power-law degree distributions. We compared our proposal with BGLL since it performed well in terms of computing time and modularity. To evaluate the effectiveness, we used synthetic graphs produced by the graph generator DIGG.

Figure 3.2 shows the computation times of our proposal and BGLL for different α values (from 2.0 to 2.5) of graphs with 1 million nodes; α represents the skewness of the power-law degree distribution. It is known that the clustering coefficient also follows a power-law degree distribution [108]; graphs with large α tend to have high clustering coefficients. As shown in Figure 3.2, BGLL shows almost constant computational time under all conditions examined. In contrast to BGLL, our algorithm increases its clustering speed as α increases. In the most efficient case, *i.e.* $\alpha = 2.5$, our proposal is up to two times faster than the result of $\alpha = 2.0$. This is because our algorithm eliminates a significant number of nodes/edges as shown in Lemma 3 and 5, when the graph has large α. Thus, our algorithm outperforms BGLL at high α values.

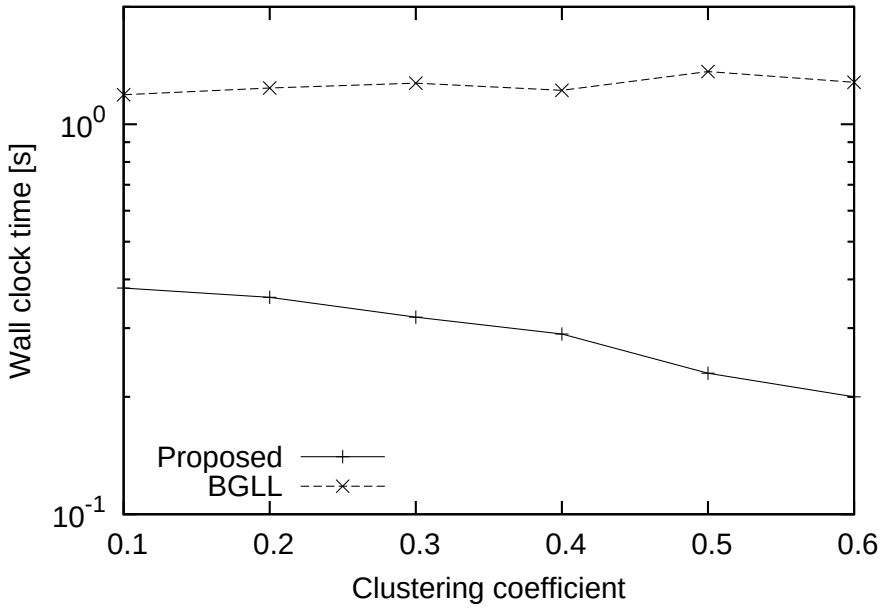

Figure 3.3: Clustering coefficient differences

We evaluated the effectiveness of our algorithm in terms of high clustering coefficients. We generated LFR benchmark graphs with 1 million nodes; the average clustering coefficient was varied from 0.1 to 0.6 following the real-world datasets. The other parameters, average degree and maximum degree were fixed at 20 and 50, respectively. Figure 3.3 shows the computation time of our proposal and BGLL for different clustering coefficient scores. As shown in Figure 3.3, BGLL shows almost constant computation time under all conditions examined. Unlike BGLL, our algorithm increased its clustering speed as the clustering coefficient increased. In the most efficient case (*i.e.* clustering coefficient is 0.6) our algorithm was up to 1.9 times faster than the result of the worst case (*i.e* clustering coefficient is 0.1). These results imply that our incremental agglomerative algorithm effectively prunes the computations for the graphs with high clustering coefficient.

Figure 5.7 shows the scalability for our proposal and BGLL; we show the wall clock time as a function of the number of nodes. We varied the number of nodes from 10 thousand to 100 million with $\alpha = 2.5$. As shown in Figure 5.7, our algorithm scales better than BGLL. This is because we do not traverse all nodes/edges multiple times. Thus, our proposal clearly achieves higher scalability than BGLL.

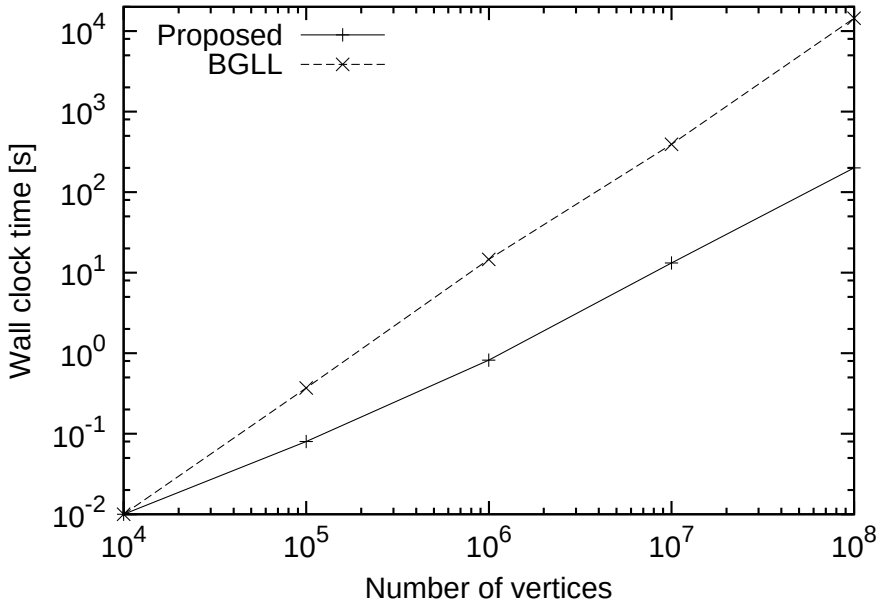

Figure 3.4: Scalability of each algorithm

3.5 Summary of this Chapter

We have introduced an efficient algorithm for finding clusters with high modularity that allows graphs of unprecedented size to be processed in practical time. Our algorithm is based on three ideas. First, it incrementally aggregates nodes, which are placed in a same cluster, into a single node. Second, it incrementally prunes computations for nodes whose clusters can be obtained. Last, it dynamically selects the node with the smallest degree. Experiments show that our algorithm can achieve efficient clustering with high modularity. Modularity-based algorithms are fundamental to many current and prospective applications in various disciplines.

Chapter 4

Parallel algorithm for Modularity-based Graph Clustering

As we described in Chapter 3, the state-of-the-art algorithm BGLL has to compute modularity gains of all adjacent node pairs to find clusters. However, this demands extremely high computation time for large-scale graphs. For this reason, we proposed IMAC (incremental modularity agglomerative clustering) in the previous chapter. To supplement the previous chapter, we propose here another variation of efficient modularity-based clustering, named *ParBGLL*, by using a data parallel computation scheme.

4.1 Introduction

As shown in the previous Chapter, the state-of-the-art algorithm BGLL is very inefficient at finding clusters in large-scale graphs. In Chapter 3, we proposed an efficient clustering algorithm, named IMAC, from the algorithmic perspective.

In this chapter, in order to overcome the performance limitation of BGLL, we tackle efficient clustering from the parallel computation and implementation perspective. Our solution is the novel data parallel graph clustering algorithm, named ParBGLL. ParBGLL is based on two ideas: First, in order to reduce the computation cost for node references, we adopt a graph data representation for BGLL that is CPU cache efficient. Specifically, we employ *compressed row storage* (CRS format

for short) [39] instead of the adjacency list representation. Our algorithm extends the CRS format to represent both adjacency lists and degrees, and increases the efficiency of CPU prefetch for fast clustering. Second, in order to reduce the computation time for modularity gain computations, we employ a data parallel method with Streaming SIMD Extensions (SSE), which is the SIMD instruction set extension of the x86. SIMD instructions are the extended instructions included in most modern CPU designs in order to improve the performance of multimedia applications. They perform the same operation on multiple data points simultaneously, and exploit data level parallelism in the CPU core but not concurrency. Since, real-world graphs have highly skewed degree distributions, there are a lot of small degree nodes. By computing these small degree nodes efficiently by using SIMD instructions, we can dramatically improve the clustering speed for large-scale graphs.

Our proposal have three major advantages:

- **Efficiency:** Our approach successfully overcomes the bottlenecks of the state-of-the-art method BGLL; the extended CRS format increases the cache hit ratio and the data parallel method with SIMD instructions is very efficient at performing the modularity gain computations.
- **High modularity:** Our evaluations reveal that the clustering results produced by our approach achieve almost same modularity scores as the state-of-the-art method BGLL.
- **Scalability:** Our proposal has better scalability than the state-of-the-art approach BGLL. Our approach reasonably increases its clustering speed according to the level of parallelism for large-scale graphs.

4.2 Preliminary

In this section, we formally define the notations and introduce the background of this chapter. Table 4.1 lists the main symbols and their definitions that are newly used in this chapter.

Prior to discussing the details of our proposal, we first introduce the SIMD instructions that are the key technique of our proposed method ParBGLL.

Table 4.1: Definition of main symbols

Symbol	Definition
V_a	node array of CRS-based graph representation given in Section 4.3.2
E_a	edge array of CRS-based graph representation given in Section 4.3.2
$\triangle Q'_{ij}$	Relative modularity gain between cluster i and j defined by Definition 12
p	maximum parallelism of SIMD-based computation

Single Instruction Multiple Data (SIMD) is a term proposed by Flynn in 1966 [113] and it means that a single instruction set is applied to multiple data items. In practice, the cores in modern processors have functional units that perform the same operation on multiple data items simultaneously. These units are called SIMD units, and instructions based on SIMD units are called SIMD instructions. Today's processors have 128-bit wide SIMD instructions that can, for example, perform operations on four 32 bit elements simultaneously. For instance, the Intel Core i7 processor is a quad-core processor with 128-bit SIMD(SSE), and NVIDIA GeForce GTX 280 GPU is a 30-core processor with 256-bit SIMD.

Recently, many approaches have been proposed that use SIMD units to perform tree search [114, 115]. However, to the best of our knowledge, there are no graph clustering approaches that have been optimized for SIMD units/instructions. Therefore, we tackled this challenging problem in developing a SIMD-based graph clustering algorithm.

4.3 Proposed method

This section overviews our proposal. The main objective is to develop a data parallelism technique for modularity-based graph clustering on modern CPUs. Our data parallel method is based on SIMD instructions. The heart of our solution is to compute the modularity gain between nodes, which takes the greatest part of the computing time, in parallel by transforming the modularity's formula into simple representations. Moreover, in implementing a clustering algorithm, we introduce a data structure that improves the cache hit ratio of node selection for greater computing efficiency. In the following sections, we first give an overview the ideas underlying our algorithm followed by a full description including the graph clustering technique.

4.3.1 Ideas

The proposed method consists of two building blocks. First, in order to reduce computation cost for node references, we adopt CPU cache and prefetch efficient graph data representation for BGLL. Specifically, we employ *compressed row storage* (CRS format for short) [39] instead of the adjacency list representation. CRS format is one of the most popular storage formats for sparse matrixes. It puts subsequent nonzero elements of the sparse matrix rows in contiguous memory locations. Hence, the format enables us to increase the efficiency of node referencing. Our algorithm extends the CRS format to represent both adjacency lists and degrees, which increases the cache hit ratio for efficient clustering.

Second, in order to reduce the computation time for modularity gain computations, we employ a data parallel method with Streaming SIMD Extensions (SSE), which is the SIMD instruction set extension of the x86. SIMD instructions are the extended instructions included in most modern CPU designs in order to improve the performance of multimedia applications. They perform the same operation on multiple data points simultaneously, and exploit data level parallelism on the CPU core but not concurrency. Hence, our algorithm uses SIMD instructions in computing the modularity gain in order to increase clustering speed. Furthermore, for efficient computation, we reduce the number of instructions that are required for each modularity gain computation by transforming the modularity's formula into simple representations.

These approaches have three major advantages. The first advantage is that only small computation cost is incurred in extracting clusters from large-scale graphs. Our approach successfully overcomes the bottlenecks of the state-of-the-art method BGLL; the extended CRS format increases the cache hit ratio and the data parallel method with SIMD instructions can efficiently perform the modularity gain computations. The second advantage is that our proposal has better scalability than the state-of-the-art approach BGLL. Our approach reasonably increases its clustering speed with the parallelism level for large-scale graphs. The third advantage is the high modularity of the clustering results. Our evaluations revealed that the clustering results produced by our approach achieve almost the same modularity scores as the state-of-the-art method BGLL.

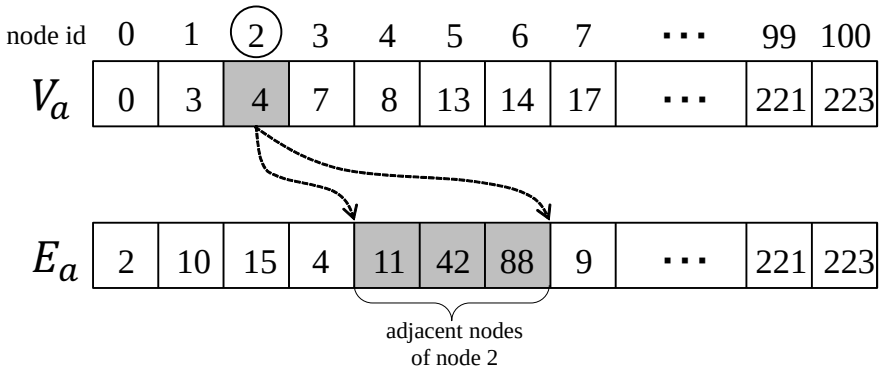

Figure 4.1: CRS-based graph representation

4.3.2 CRS-based Graph Representation

Graph $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$ is commonly represented as an adjacency matrix. For sparse graphs such a representation wastes a lot of space. Furthermore, the matrix representation also increases data access costs for clustering since state-of-the-art algorithms iteratively access all nodes and edges for finding clusters and these computations on the matrix representation decrease cache and prefetch efficiency.

In order to overcome the above limitations, we present graphs in compact adjacency list form that are packed into large arrays called CRS [39]. We introduce here details of the CRS-based graph representation. This graph representation consists of two arrays: node array V_a and edge array E_a. Each node points to starting position of its own adjacency list in this large array of edges. Figure 4.1 shows an example of CRS-based data representation. Each array is formally given as follows:

- **Node array:** Nodes of the graph are represented as an linear array, called node array V_a. Each index and entry of node array V_a correspond to the node id and the starting index of its adjacency list in edge array E_a, respectively. For example, in Figure 4.1, node array V_a shows the starting index of its adjacency list in E_a from node 0 to node 100. Specifically, node 2 in node array V_a of Figure 4.1 has the entry of 4, and it implies that the adjacency list of node 2 starts from index 4 of E_a.
- **Edge array:** Edges of the graph are represented as a linear array, called edge array E_a. Each entry of edge array E_a refers to a node in node array V_a.

While the above graph representations use linear arrays, we can obtain adjacent nodes of a node efficiently. As described above, the node array has the starting index of its adjacency list in edge array E_a. Hence, we can obtain the adjacent nodes of a node by retrieving the indices of E_a from $V_a[u]$ to $V_a[u+1]$. For example, we can obtain the adjacent nodes of node 2, $\{11, 42, 88\}$, by retrieving E_a from $V_a[2] = 4$ to $V_a[2+1] = 7$.

Additionally, we reduce the data access costs for clustering by using the above two arrays for clustering. This is because the node array and edge array are stored in continuous memory address space. Therefore, this representation can improve prefetch efficiency for graph clustering.

4.3.3 Proposed method: ParBGLL

In this section, we introduce ParBGLL, vectorized modularity-based graph clustering that uses SIMD instructions.

Vectorization of the modularity computation

The state-of-the-art algorithm BGLL computes modularity gains for all adjacent nodes of each node. In our proposal ParBGLL, we introduce the vectorized computation form by using SIMD instructions: First, as is true for BGLL, ParBGLL selects a node to initiate the modularity gain computation. ParBGLL then obtains all adjacent nodes from the CRS-based graph representation. Last, ParBGLL performs vectorized modularity gain computation for all adjacent nodes.

In order to obtain the modularity gain, we have to evaluate the equation given by Definition 7 for each adjacent node. In this case, if we assume that we have already obtained the score of each term in Definition 7 such as e_{ij}, $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik}$ and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}$, we have to use SIMD instructions to evaluate at least the following six instructions:

- quotient of e_{ik} and $2m$
- quotient of $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik}$ and $2m$
- quotient of $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}$ and $2m$

- product of $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik}/2m$ and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}/2m$
- difference between $e_{ij}/2m$ and $(\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik}/2m)(\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}/2m)$
- a product between 2 and $\{e_{ij}/2m - (\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik}/2m)(\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}/2m)\}$

In order to reduce the number of instructions, we focus on the locality of graph clustering computations in real-world graphs. In the local clustering phase of BGLL, we have to find the adjacent node that maximizes the modularity gain among all adjacent nodes of the node. Traditionally, the original algorithm BGLL computes the full description of Definition 7, hence it requires at least six instructions for each computation as we described above. However, almost modularity gain computations have similar mathematical expressions since BGLL only needs to select an adjacent nodes whose modularity gain is local maximum. More specifically, if we compute the modularity gains for the adjacent nodes of node u, all the modularity gain computations have same values of $2m$ and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk}$: They differ only in values of e_{uj} and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}$. In fact, we can compute the relative size of the modualrity gain by only comparing e_{uj} and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}$; and this relative form reduces the number of instructions for each modularity computation. Hence in our proposal, we only compute relative form of the modularity gain, defined in Definition 12, instead of the full description of Definition 7. We define relative modularity gain as follows:

Definition 12 (Relative modularity gain $\triangle Q'_{ij}$) *Let m be the total number of edges and e_ij be the total number of edges between cluster $\mathbb{C}_i$ and $\mathbb{C}_j$. The following equation gives the relative modularity gain $\triangle Q'_{ij}$ among the nodes in $\Gamma(u)$:*

$$\triangle Q'_{ij} = 2me_{ij} - \sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik} \sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}. \tag{4.1}$$

From Definition 12, we can introduce the following Lemma:

Lemma 6 (Equivalence between $\triangle Q_{ij}$ and $\triangle Q'_{ij}$) *Let node v, w be the adjacent nodes of node u. We always have,*

$$\triangle Q_{u,v} > \triangle Q_{u,w} \Leftrightarrow \triangle Q'_{u,v} > \triangle Q'_{u,w}. \tag{4.2}$$

Proof We first prove the sufficient condition of Lemma 6. From Definition 7, we

have

$$\triangle Q_{uv} = 2\left\{\frac{e_{uv}}{2m} - \left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk}}{2m}\right)\left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{vk}}{2m}\right)\right\} \quad (4.3)$$

$$\triangle Q_{uw} = 2\left\{\frac{e_{uw}}{2m} - \left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk}}{2m}\right)\left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{wk}}{2m}\right)\right\} \quad (4.4)$$

Therefore, the following holds,

$$\triangle Q_{uv} > \triangle Q_{uw} \quad (4.5)$$

$$2\left\{\frac{e_{uv}}{2m} - \left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk}}{2m}\right)\left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{vk}}{2m}\right)\right\} > 2\left\{\frac{e_{uw}}{2m} - \left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk}}{2m}\right)\left(\frac{\sum_{\mathbb{C}_k \in \mathbb{C}} e_{wk}}{2m}\right)\right\} \quad (4.6)$$

$$2me_{uv} - \sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk} \sum_{\mathbb{C}_k \in \mathbb{C}} e_{vk} > 2me_{uw} - \sum_{\mathbb{C}_k \in \mathbb{C}} e_{uk} \sum_{\mathbb{C}_k \in \mathbb{C}} e_{wk} \quad (4.7)$$

$$\triangle Q'_{u,v} > \triangle Q'_{u,w}. \quad (4.8)$$

As a result, we have $\triangle Q_{u,v} > \triangle Q_{u,w} \Rightarrow \triangle Q'_{u,v} > \triangle Q'_{u,w}$ The necessary condition of Lemma 6 can prove the same approach that we show above. Therefore, we hold $\triangle Q_{u,v} > \triangle Q_{u,w} \Leftrightarrow \triangle Q'_{u,v} > \triangle Q'_{u,w}$ □

As we can see, the relative modularity gain contains only three instructions as follows:

- product of e_{ij} and $2m$
- product of $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik}$ and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}$
- difference between $2me_{ij}$ and $\sum_{\mathbb{C}_k \in \mathbb{C}} e_{ik} \sum_{\mathbb{C}_k \in \mathbb{C}} e_{jk}$

Hence, we can reduce the number of instructions by using the relative modularity gain instead of the modularity gain given by Definition 7.

ParBGLL uses the above relative modularity gain to find clusters in a data parallel manner. Figure 4.2 shows our vectorized modularity gain computations that are based on the relative modularity gain of Definition 12. As shown in Figure 4.2, when we compute the relative modularity gains for the adjacent nodes, we can evaluate them in parallel provided they occupy the same SIMD register. For example, in Figure 4.2, we compute the relative modularity gains $\triangle Q'_{12}, \triangle Q'_{13}, \triangle Q'_{14}$ and $\triangle Q'_{15}$. In this case, we can evaluate them in parallel by using SIMD instructions (*i.e.* _mm_set_epi32 and _mm_sub_epi32) since these four relative modularity gain computations can occupy the same SIMD register (*i.e.* Register A in Figure 4.2).

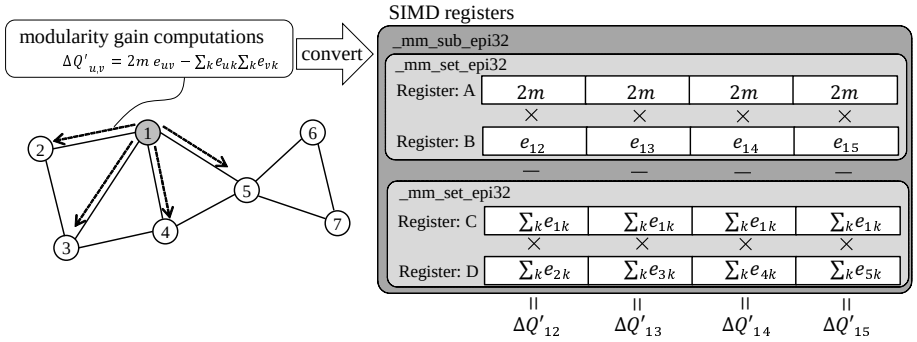

Figure 4.2: Vectorized relative modularity gain computations

Vectorized selection of the maximum modularity gain node

After computing the relative modularity gains for all adjacent nodes of a node, we then have to select the one that has the largest relative modularity gain among all adjacent nodes. We also implement this procedure by SIMD instructions.

In order to find the largest (relative) modularity gain node, we first set up five SIMD registers, named *reg_id*, *reg_result*, *reg_max_id*, *reg_max* and *reg_mask*. *reg_id* is a SIMD register that stores the adjacent node ids. *reg_result* stores values of the relative modularity gains that correspond to the *reg_id*. *reg_max_id* contains adjacent node ids that may have the largest modularity gain among the adjacent nodes. *reg_max* stores relative modularity gain values that correspond to the *reg_max_id*. Finally, *reg_mask* is a mask SIMD register that is used to select the node with the largest modularity gain.

Here, we introduce a concrete procedure for selecting the largest modularity gain node. As shown in the previous section, *reg_id* and *reg_result* are filled by subsets of the adjacent nodes and their relative modularity gain values, respectively. If *reg_max_id* and *reg_max* are empty, ParBGLL copies the entries of *reg_id* and *reg_result* to *reg_max_id* and *reg_max*, respectively. Otherwise, it updates *reg_max_id* and *reg_max*. In order to update the *reg_max_id* and *reg_max*, we first build *reg_mask* from *reg_result* and *reg_max*. For building *reg_mask*, ParBGLL compares the relative modularity gain values for each alignment between *reg_result* and*reg_max*. In each alignment, if the relative modularity value in *reg_result* is greater than that of *reg_max*, ParBGLL fills the corresponding alignment of *reg_mask* with 1. Otherwise, it fills the corresponding alignment with 0.

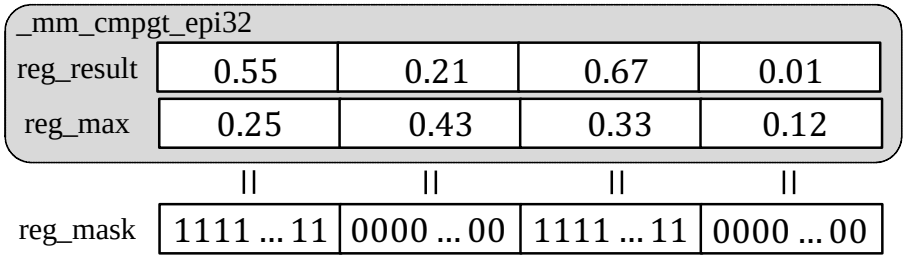

Figure 4.3: Example of *reg_mask* construction from *reg_result* and *reg_max*

Figure 4.3 shows an example of *reg_mask* construction. As we can see, the left most alignment of *reg_result* is greater than that of *reg_max*. This implies that the corresponding alignment should be updated by the result of *reg_result*. Therefore, we fill the left most alignment of *reg_mast* with 1. On the other hand, the right most alignment of *reg_result* is smaller than that of *reg_max*. This implies that this alignment does not need to be updated, and hence we fill the right most alignment with 0.

After computing *reg_mask*, ParBGLL extracts the adjacent nodes that may have the largest relative modularity gain by using *reg_mask* and *reg_max_id* as follows:

$$reg\_max\_id = (!reg\_mask \ \& \ reg\_max\_id) \ | \ (reg\_mask \ \& \ reg\_id) \tag{4.9}$$

ParBGLL continues the above procedure until all adjacent nodes of a node have been evaluated, and it continues saving and updating both *reg_max_id* and *reg_max*. After terminating the above procedure, we have the candidate adjacent nodes in *reg_max_id* that have the largest relative modularity gains. Hence, by traversing the candidate nodes in *reg_max_id* with their relative modularity gain values in *reg_max*, ParBGLL finds the largest relative modularity gain nodes from adjacent nodes.

4.3.4 Graph Clustering Algorithm

Algorithm 3 shows our algorithm. Basically, Algorithm 3 follows the algorithm of BGLL, see in Algorithm 1. First, ParBGLL fills SIMD register *reg_2m* with $2m$, repeatedly used in ParBGLL (line 2). Then, it selects a node from $\mathbb{V}$ and computes the relative modularity gain values for all adjacent nodes of the node by using our vectorized method (line 4-15). In these relative modularity gain computations, ParBGLL first prepares SIMD register *reg_u* (line 5). ParBGLL then computes

Algorithm 3 ParBGLL

Input: $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$;
Output: clustering result of $\mathbb{G}$;

1: **repeat**
2: fill SIMD register *reg_2m* with $2m$;
3: **repeat**
4: select node u from $\mathbb{V}$;
5: fill SIMD register *reg_u* with $\sum_{k \in \mathbb{C}} e_{uk}$;
6: $i = 0, \mathbb{N} = \Gamma(u)$;
7: **while** $i \leq (|\Gamma(u)|/p + 1)$ **do**
8: pop p nodes $\{v_1, v_2, \ldots, v_p\}$ from $\mathbb{N}$;
9: fill SIMD register *reg_id* with $\{v_1, v_2, \ldots, v_p\}$;
10: fill SIMD register *reg_e* with $\{e_{uv_1}, e_{uv_2}, \ldots, e_{uv_p}\}$;
11: fill SIMD register *reg_v* with $\{\sum_{k \in \mathbb{C}} e_{v_1 k}, \sum_{k \in \mathbb{C}} e_{v_1 k}, \ldots, \sum_{k \in \mathbb{C}} e_{v_1 k}\}$;
12: compute *reg_result* $=$ (*reg_2m*)(*reg_e*) $-$ (*reg_u*)(*reg_v*);
13: get *reg_mask* from *reg_result* and *reg_max*;
14: get *reg_max* and *reg_max_id* by Eq. (4.9);
15: i++;
16: **end while**
17: search for node v' that has the largest $\triangle Q'_{uv'}$ from *reg_max* and *reg_max_id*;
18: **if** the modularity gain of node $v' > 0$ **then**
19: assign node u to the cluster of node v';
20: **else**
21: keep node u in its original cluster;
22: **end if**
23: **until** modularity Q is increased
24: aggregate clusters into a weighted graph;
25: **until** modularity Q is increased

the modularity gain values for all adjacent nodes of node u in data parallel manner. In Algorithm 3, we assume the maximum parallelism equals p. Therefore, in each data parallel computation, ParBGLL picks p nodes from $\Gamma(u)$ (line 8), and computes gains by using SIMD instructions (line 9-15). These data parallel computations continue until there are no unevaluated nodes in $\Gamma(u)$. After the modularity gain computations, ParBGLL selects adjacent node v' that maximizes the modularity gain from *reg_max_id* and *reg_max* (line 17), and assigns node u and v' to the same cluster if the gain between them is positive (line 18-22). The consequent procedures are same as those of BGLL (line 24).

We provide here a theoretical analysis of the computation cost of ParBGLL

Theorem 2 *(Computation cost) ParBGLL requires $O(\frac{1}{p}|\mathbb{E}| + |\mathbb{V}|)$ time to cluster a graph, where p, $|\mathbb{E}|$ and $|\mathbb{V}|$ are the maximum parallelism level, the number of edges, and the number of nodes, respectively.*

Proof Since the average degree is $|\mathbb{E}|/|\mathbb{V}|$, the number of iterations consumed by Algorithm 3 (line 7-16) for each node, is $\frac{1}{p}|\mathbb{E}| + 1$. Our algorithm has to iterate the

Table 4.2: Real-world datasets

	dblp-2010	ljournal-2008	uk-2005	webbase-2001	uk-2007-05
$\|\mathbb{V}\|$	326,186	5,363,260	39,459,925	118,142,155	105,896,555
$\|\mathbb{E}\|$	1,615,400	79,023,142	936,364,282	1,019,903,190	3,738,733,648
α	2.82	2.29	1.71	2.14	1.51

above computations at least $|\mathbb{V}|$ times. Therefore, the computation cost of ParBGLL is $O(\frac{1}{p}|\mathbb{E}| + |\mathbb{V}|)$. □

4.4 Experimental evaluation

We conducted evaluations to confirm the effectiveness of our algorithm. In the experiments, we used the following five public datasets [111] to evaluate our algorithm:

- *dblp-2010*: This scientific collaboration graph was extracted from the bibliography service DBLP in 2010; each node is a scientist and each edge is coauthor relationship.
- *ljournal-2008*: This graph was obtained from the social networking site LiveJournal in 2008; each node and edge represent a user and friendship among users, respectively.
- *uk-2005*: This graph was obtained from a 2005 crawl of .uk domain; each node and edge indicate a Web page and a link between pages, respectively.
- *webbase-2001*: This Web graph of .us domain in 2001 was downloaded from the Stanford Webbase project, each node and edge represent a Web page and a link, respectively.
- *uk-2007-05*: This graph is a expansion of *uk-2005* crawled from .uk domain Web pages in May, 2007.

The details of our datasets are shown in Table 5.2, where α is the exponent that controls the skewness of the degree distribution, described in the previous section. Additionally, we also used synthetic datasets generated by DIGG[1] as same as the

previous section. The specific parameter settings are given later.

All experiments were conducted on a Linux 2.6.18 server with Intel Xeon CPU L5640 2.27GHz and 144GB RAM. In our evaluation, the maximum parallelism is 4 since we used SIMD register size and node unit size of 128 bits and 32 bits, respectively. We implemented our proposal using C++ and we used the optimization parameter "-O2" for each algorithm. To evaluate the existing algorithms, we used BGLL programs published on the authors' sites.

4.4.1 Efficiency

We evaluated the clustering performance of each algorithm through wall clock time for each real world dataset. Figure 4.4 shows the computation times recorded. For Figure 4.4, our proposal was tested in two variants; *Cache+SIMD* and *Cache*. Cache+SIMD represents the full version of our algorithm that uses the cache efficient data structure and the SIMD-based parallelism. Cache is the limited version of proposal that does not uses SIMD-based parallelism. Figure 4.4 indicates that Cache and Cache+SIMD are superior to the original algorithm BGLL under all experimental conditions. Specifically, Cache is at least five times faster than BGLL, and Cache+SIMD is almost 20 times faster than BGLL. As described earlier, the existing algorithms traverse all nodes/edges multiple times and this computation decreases the prefetch efficiency. In contrast, Cache and Cache+SIMD employ CRS based data layout for clustering. Therefore, both of Cache and Cache+SIMD can increase the prefetch efficiency and the clustering speed.

In order to experimentally verify the prefetch efficiency of our proposal, we evaluated the CPU L2/L3 cache hit ratio for Cache and BGLL. To monitoring the CPU cache hit ratio, we used Intel® Performance Counter Monitor [116]. Figure 4.5 and 4.6 show that the evaluation result of L2 and L3 cache, respectively. As we can see, Cache shows higher L2 and L3 cache hit ratios for large-scale datasets such as uk-2005, webbase-2001 and uk-2007-05. This implies that our CRS-based graph representation improves the prefetch efficiency and as a result, the L2 and L3 cache hit ratios are increased. Figure 4.5 and 4.6 also show that the CRS-based representation matches BGLL for smaller datasets such as dblp-2010. Although the L3 cache size of our experimental environment is 12 MB, dblp-2010 only consumes roughly 6MB for clustering. Therefore, our approach is also competitive for smaller

[1] http://digg.cs.tufts.edu/

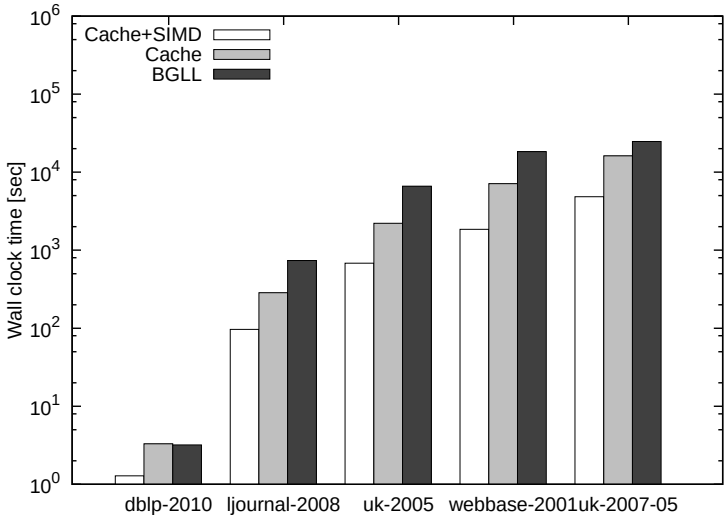

Figure 4.4: Running time for real-world datasets

datasets. Figure 4.4 also indicates that Cache+SIMD is almost four times faster than Cache for all datasets. As described above, we set the maximum parallelism of SIMD-based computation to 4 in this evaluation. The results of Figure 4.4 imply that our method offers reasonable performance. More specifically, we can see that the performance improvement of Cache+SIMD is high for datasets with high α values such as dblp-2010, ljournal-2008, and webbase-2001. The reason is simple; these datasets have a lot of small degree nodes since their degree distributions are highly skewed. As we can see in Algorithm 3, ParBGLL can compute the relative modularity gains for all adjacent nodes by a single iteration if node degree is smaller than the level of parallelism p. Therefore, if a graph has a highly skewed degree distribution that follows a power-law, our algorithm offers improved clustering performance.

4.4.2 Modularity

We evaluated the clustering quality of Cache, Cache+SIMD and BGLL in terms of modularity. Table 4.3 shows modularity Q values for each of the real world datasets. As shown in Table 4.3, our evaluation revealed that our proposal produces almost same clustering results in terms of modularity as BGLL. This is because the relative modularity gain given by Definition 12 can always find the same node

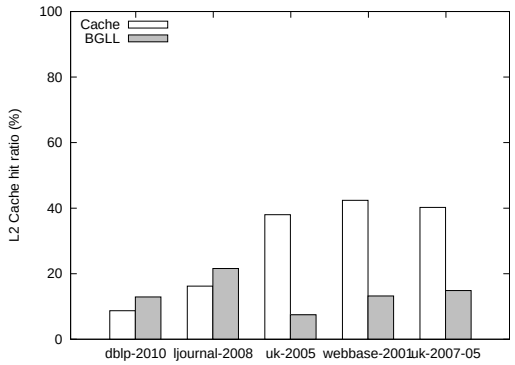

Figure 4.5: L2 cache hit ratio

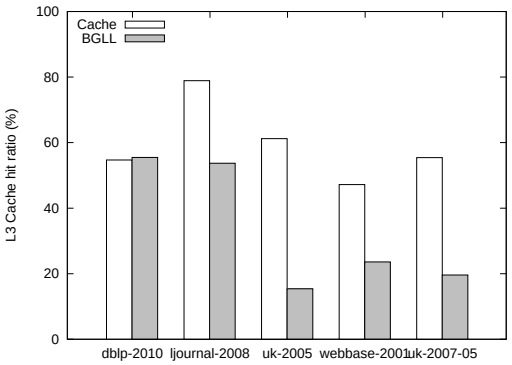

Figure 4.6: L3 cache hit ratio

Table 4.3: Modularity Q for each dataset

	dblp-2010	ljournal-2008	uk-2005	webbase-2001	uk-2007-05
Cache+SIMD	0.868	0.755	0.977	0.980	0.982
BGLL	0.871	0.754	0.979	0.981	0.979

that maximizes the original form of modularity gain (Lemma 6). Hence, clustering quality equals that of BGLL even though Figure 4.4 shows our algorithms are much faster than BGLL.

4.4.3 Scalability

We evaluate the scalability of our proposal by using synthetic datasets produced by the graph generator DIGG. We set the parameter α, which controls the skewness of the power-law degree distribution, as $\alpha = 2.0$. Under the above condition, we produced four synthetic graphs with 10 thousand, 100 thousand, one million, and 10 million nodes. Figure 4.7 shows the wall clock time of clustering for the four synthetic datasets. As shown in Figure 4.7, our proposal is only up to two times faster than BGLL when graph size is small (*i.e.* 10 thousand and 100 thousand nodes). In contrast, the clustering speed of our algorithm is increased when graph size is large (*i.e.* one million and 10 million nodes).

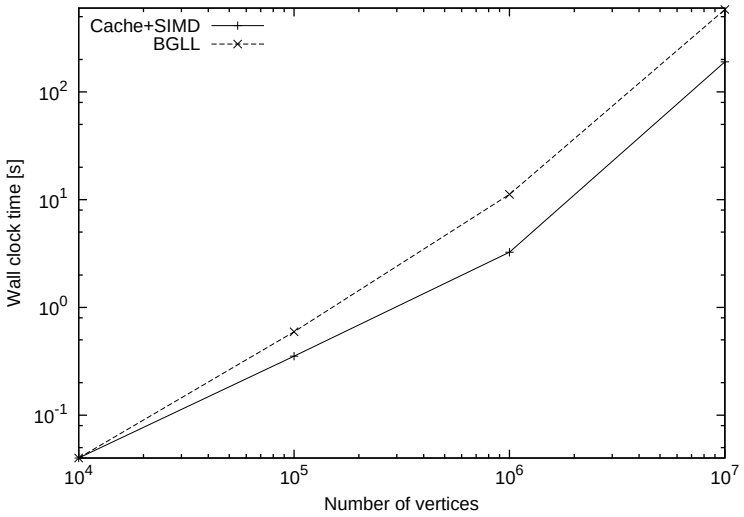

Figure 4.7: Scalability for each method

4.5 Summary of this Chapter

In this chapter, we have introduced a vectorized modularity-based clustering algorithm named ParBGLL by extending the state-of-the-art algorithm BGLL. Our algorithm is based on two ideas. First, we extend the CRS format to produce a data layout that suits graph clustering by increasing cache and prefetch efficiency. Second, we convert the modularity gain computation into a vectorized computation form by using Streaming SIMD Extensions (SSE), the instruction set extension for the x86 CPU. Experiments showed that our algorithm can achieve scalable clustering with high modularity.

Chapter 5

Efficient Algorithm for Structural Clustering

The structural clustering algorithm *SCAN*, proposed by Xu *et al.* [41], is successfully used in many applications because it not only detects densely connected nodes as clusters but also identifies sparsely connected nodes as hubs or outliers. However, it is difficult to apply SCAN to large-scale graphs due to its high time complexity. In this chapter, we propose a novel graph clustering algorithm named *SCAN++*. In order to reduce time complexity, we introduce new data structure of *directly two-hop-away reachable node set (DTAR)*. DTAR is the set of two-hop-away nodes from a given node that are likely to be in the same cluster as the given node. SCAN++ employs two approaches for efficient clustering by using DTARs without sacrificing clustering quality. As a result, SCAN++ detects exactly the same clusters, hubs, and outliers from large-scale graphs as SCAN with much smaller computation time.

5.1 Introduction

Besides extracting clusters, finding special role nodes, *hubs* and *outliers*, is also a worthwhile task for understanding the structures of large-scale graphs [117]. Hubs are generally thought of as bridging different clusters. In the context of graph data mining, they are often considered as representative or influential nodes. In contrast, outliers are the nodes that are neither clusters nor hubs; they are treated as noises.

The hubs and outliers provide useful insights in mining graphs. For instance, hubs in web graphs act like authoritative web pages that link similar topics [118]. The detection of outliers in web graphs is useful in stripping spam pages from web pages [119]. As well as web analysis, hubs and outliers play important roles in various applications such as marketing [120] and epidemiology [62]. That is why identifying hubs and outliers has become an interesting and important problem.

Most traditional clustering algorithms such as graph partitioning [85, 90, 121, 122], modularity-based clustering [34, 36, 38, 123], density-based clustering [124–126], and clique detection [127, 128] only study the problem of cluster detection and so ignore hubs and outliers. One of the most successful clustering methods is *structural clustering algorithm* (*SCAN*) proposed by Xu *et al.* [41]. Similar to density-based clustering and clique detection, the main concept of SCAN is that densely connected adjacent nodes should be in the same cluster. However, unlike the traditional algorithms, SCAN successfully finds, at not insignificant cost, not only clusters but also hubs and outliers. As a result, it has been used in many applications in social, web and bioinfomatics domains.

Although SCAN's effectiveness in detecting hubs and outliers as well as clusters is known in many applications, SCAN has, unfortunately, a serious weakness; it exponentially increases its computation cost with the number of edges. This is because SCAN has to find all clusters prior to identifying hubs and outliers; it first finds densely connected node sets as clusters. It then classifies the remaining non-clustered nodes into hubs or outliers. This clustering procedure entails exhaustive density evaluations for all adjacent nodes in large-scale graphs. Several methods have been proposed to improve its clustering speed. For example, LinkSCAN*, proposed by Lim *et al.*, is a state-of-the-art algorithm that uses SCAN to find overlapping communities from large-scale graphs [104]. They improve the efficiency of SCAN by employing an *edge sampling technique* [104] in the clustering process. By sampling edges from the graphs, they reduce the number of edges that require density evaluations. However, this approach produces approximated clustering results; it does not guarantee the same clustering results as the original algorithm, and so loses the superiority of SCAN [104].

We present a novel algorithm, *SCAN++*, which mitigates the low efficiency of SCAN. Our proposal can efficiently handle graphs with more than several million nodes and edges. Furthermore, SCAN++ guarantees the exactness of its clustering result; SCAN++ always returns exactly the same clusters, hubs and outliers as SCAN.

SCAN++ is based on the property of real-world graphs; real-world graphs such as web graphs have high scores of *clustering coefficients* [3, 108]. The clustering coefficient of a node is a measure of node density. If a node and its neighbor nodes approach a complete graph (*a.k.a.* a clique), the score of clustering coefficient becomes high. That is, a node and its two-hop-away nodes especially in real-world graphs are expected to share large parts of their neighborhoods. Based on this property, SCAN++ prunes the density evaluation for the nodes that are shared between a node and its two-hop-away nodes. Specifically, SCAN++ employs the following techniques: (1) it uses a new data structure, *directly two-hop-away reachable node set (DTAR)*, the set of nodes that are two hops away from a given node, (2) it reduces the cost of clustering by avoiding unnecessary density evaluations if the nodes are not included in DTARs, and (3) its density evaluation is efficient since DTAR allows the reusing of density evaluation results. After identifying clusters, SCAN++ classifies the remaining nodes, which do not belong to clusters, as hubs or outliers. Instead of the exhaustive computation performed by the original algorithm, SCAN++ can find clusters in an efficient manner in large-scale real-world graphs.

SCAN++ has the following attractive characteristics:

- **Efficient**: SCAN++ exploits DTAR to avoid exhaustively evaluating the density of all adjacent nodes in the graphs (Section 5.3.3 and 5.3.4). Consequently, it achieves higher clustering speeds than SCAN as well as the edge sampling technique of LinkSCAN* (Section 5.5.2). Although SCAN requires exponential clustering time against the number of edges, SCAN++ has near-linear clustering time against the number of edges (Section 5.4.1 and 5.5.5).
- **Exact**: Unlike state-of-the-art algorithms, SCAN++ theoretically guarantees the same clustering results as SCAN, even though it drops unnecessary density evaluations (Section 5.4.2). As a result, SCAN++ always returns exactly same clusters, hubs and outliers as original SCAN (Section 5.5.3).
- **Effective**: As described above, real-world graphs have high scores in terms of clustering coefficients. Accordingly, we designed SCAN++ to effectively handle graphs of high clustering coefficients by using DTAR. Hence, SCAN++ offers efficient clustering for large-scale real-world graphs that exhibit high clustering coefficients (Section 5.5.4).

To the best of our knowledge, SCAN++ is the first solution to achieve both high efficiency and clustering results guarantees at the same time. Our experiments confirm that SCAN++ computes clusters, hubs and outliers 20 times faster than original

Table 5.1: Definition of main symbols

Symbol	Definition
ϵ	Threshold of the structural similarity, $0 \leq \epsilon \leq 1$
μ	Minimal number of nodes in a cluster
$\mathbb{H}$	Set of hubs in $\mathbb{H}$
$\mathbb{O}$	Set of outliers in $\mathbb{G}$
$\mathbb{B}$	Set of bridges in $\mathbb{G}$
$\mathbb{N}[u]$	Set of nodes in the structure neighborhoods of node u
$\mathbb{N}_\epsilon[u]$	Set of nodes in the ϵ-neighborhoods of node u
$\mathbb{D}[u]$	Set of directly structure-reachable nodes of node u
$\mathbb{C}[u]$	Set of nodes that belong to the same cluster as node u
$\mathbb{T}[u]$	Set of nodes in the DTAR of node u
$\mathbb{T}_u$	Set of nodes in the converged DTAR of node u
$\mathbb{L}[u]$	Set of nodes in the local cluster of node u
$\mathbb{V}_{\mathbb{T}_u}$	Set of candidate nodes of clusters derived from $\mathbb{T}_u$
$\mathbb{P}_\epsilon[b]$	Set of pivots in the ϵ-neighborhood pivots of bridge b
$\sigma(u, v)$	Structural similarity between node u and v

SCAN on average without sacrificing the clustering quality. Even though original SCAN is effective in enhancing application quality, it has been difficult to apply to large-scale graphs due to its performance limitation. However, by providing a sophisticated approach that suits the identification of clusters, hubs, and outliers, SCAN++ will help to improve the effectiveness of a wider range of applications.

5.2 Preliminary

In this section, we formally define the notations and introduce the background of this chapter. Table 5.1 lists the main symbols and their definitions that are newly used in this chapter.

We briefly review the original algorithm of SCAN proposed by Xu *et al.* [41]. SCAN is one of the most popular graph clustering methods; it successfully detects not only clusters $\mathbb{C}$ but also hubs $\mathbb{H}$ and outliers $\mathbb{O}$ in the given graph unlike traditional methods. Intuitively, SCAN extracts clusters as sets of nodes that have dense internal connections; it identifies the other non-clustered nodes (*i.e.* nodes that belong to none of the clusters) as hubs or outliers. More specifically, SCAN

determines a non-clustered node as a hub if it bridges different clusters; otherwise, the node is determined as an outlier. Therefore, prior to identifying hubs and outliers, it finds all clusters in a given graph.

SCAN detects clusters based on a simple idea; if adjacent nodes are densely connected to each other, they should be assigned to the same cluster. To find clusters based on this idea, SCAN first detects a special node, called *core*. Core is a node that has a lot of neighbor nodes with highly dense connections; the core is regarded as the seed of a cluster. SCAN uses the *structural neighborhood* [41] to evaluate density. The structural neighborhood of a node is a node set composed of the node itself and all its adjacent nodes.

Definition 13 (Structural neighborhood) *The definition of structural neighborhood of node u, denoted by $\mathbb{N}[u]$, is given as follows:*

$$\mathbb{N}[u] = \{v \in \mathbb{V} : (u, v) \in \mathbb{E}\} \cup \{u\}. \tag{5.1}$$

The density of adjacent nodes is computed by the common nodes in the structural neighborhoods. SCAN measures the number of common nodes in two structural neighborhoods normalized by the geometric mean of their structural neighborhood sizes. This measurement is called *structural similarity* and is defined as follows:

Definition 14 (Structural similarity) *The following equation gives the structural similarity, denoted by $\sigma(u, v)$, between node u and v.*

$$\sigma(u, v) = \frac{|\mathbb{N}[u] \cap \mathbb{N}[v]|}{\sqrt{|\mathbb{N}[u]||\mathbb{N}[v]|}}. \tag{5.2}$$

The definition is extended from a cosine similarity, and it can be replaced by other measures such as Jaccard similarity [101]. The structural similarity is a score varying from 0 to 1 that indicates the scale of matching degree of structural neighborhoods. When adjacent nodes share many members of their structural neighborhoods, their structural similarity becomes large.

From Definition 14, SCAN detects the core by evaluating structural similarities for all neighborhoods. A node is core if the node has sufficiently large structural similarities with enough number of its adjacent nodes. In order to specify core metrics, SCAN requires two user-specified parameters. First is the minimum score of the structural similarity to neighbor nodes, denoted by ϵ. Second is the minimum

number of neighborhoods, denoted by μ, all of whose structural similarities exceed ϵ. SCAN regards a node as core when it has at least μ neighbors with structural similarities greater than ϵ:

Definition 15 (Core) *Node u is core iff the node has at least μ neighbor nodes whose structural similarities are greater than ϵ. More precisely,*

$$\textit{Node } u \textit{ is core } \Leftrightarrow |\mathbb{N}_\epsilon[u]| \geq \mu, \tag{5.3}$$

where $\mathbb{N}_\epsilon$, called ϵ-neighborhood, is defined as follows:

$$\mathbb{N}_\epsilon[u] = \{v \in \mathbb{N}[u] : \sigma(u, v) \geq \epsilon\}. \tag{5.4}$$

Once SCAN finds core, SCAN expands a cluster from the core. Specifically, nodes included in the ϵ-neighborhood of the core are assigned to the same cluster as the core. The ϵ-neighborhood nodes of the core are called *directly structure-reachable nodes*, and are defined as follows:

Definition 16 (Directly structure-reachable) *The following equation gives directly structure-reachable nodes of node u denoted by $\mathbb{D}[u]$:*

$$\mathbb{D}[u] = \begin{cases} \mathbb{N}_\epsilon[u] & (|\mathbb{N}_\epsilon[u]| \geq \mu) \\ \emptyset & (|\mathbb{N}_\epsilon[u]| < \mu) \end{cases} \tag{5.5}$$

When node u is core and $\mathbb{D}[u] \neq \emptyset$, SCAN assigns all nodes in $\mathbb{D}[u]$ to the same cluster as node u.

SCAN recursively expands the cluster by checking whether each node, which is included in the cluster, satisfies core condition defined by Definition 15 or not. Specifically, if (1) node v is included in $\mathbb{D}[u]$ and (2) node v is core, SCAN assigns nodes in $\mathbb{D}[v]$ to the same cluster as node u. These directly structure-reachable nodes (*i.e.* $\mathbb{D}[v]$) are expanded from a member node of the cluster (*i.e.* $\mathbb{D}[u]$). These expanded directly structure-reachable nodes $\mathbb{D}[v]$ are called *structure-reachable nodes* of node u. On the other hand, if (1) node v is included in $\mathbb{D}[u]$ and (2) node v is *not* core, SCAN does not expand the cluster from the node. All nodes in a cluster, except the core node, are called *border* nodes. SCAN recursively finds cores and expands the clusters from the cores until there are no undiscovered cores in the structure-reachable nodes of node u. After completion of cluster expansion, SCAN obtains the structure-reachable nodes of node u, which are composed of cores and borders. The original algorithm determines the obtained nodes as being in the same cluster as node u. Formally, the cluster that has node u is defined as follows:

Definition 17 (Cluster) *The following equation obtains the cluster by node u, denoted by $\mathbb{C}[u]$:*

$$\mathbb{C}[u] = \{w \in \mathbb{D}[v] : v \in \mathbb{C}[u]\}, \tag{5.6}$$

where $\mathbb{C}[u]$ is initially set to $\mathbb{C}[u] = \{u\}$.

Note that a cluster is uniquely determined by the cores included in the cluster [101]. So, if we find the same cores as SCAN, we also detect the same clusters as SCAN. After termination of cluster expansion, SCAN randomly selects a new node from the nodes that have yet to be checked. SCAN continues this procedure until there are no undiscovered cores.

After detecting clusters, SCAN identifies non-clustered nodes (*i.e.* nodes that belong to no cluster) as hubs or outliers. The idea is simple; a node is a hub if the node is connected to multiple clusters, otherwise it is an outlier.

Definition 18 (Hub and Outlier) *Assume node u does not belong to any cluster. $u \in \mathbb{H}$ iff node v and w exist in $\mathbb{N}[u]$ such that $\mathbb{C}[v] \neq \mathbb{C}[w]$. Otherwise $u \in \mathbb{O}$.*

Note that, as described in the literature [41], the definition of a hub and an outlier is flexible enough for practical application. For example, it may be more appropriate than the above definition for some applications to determine a non-clustered node with extremely high degree as a hub. This point should be discussed in future when we consider actual applications.

As a result, SCAN finds all clusters, hubs, and outliers in a graph. However, despite its effectiveness in finding the hidden structure of graphs, it is difficult to apply SCAN to large-scale graphs since it requires high time complexity. This is because the clustering procedure entails exhaustive similarity evaluations for all adjacent nodes in the given graph; SCAN has to traverse all nodes and compute the structural similarities for all of the adjacent nodes in the graph. Thus, if $\mathbb{V} = \{u_1, u_2, \ldots, u_{|\mathbb{V}|}\}$, the running cost of SCAN is of the order of $O(|\mathbb{N}[u_1]| + |\mathbb{N}[u_2]| + \cdots + |\mathbb{N}[u_{|\mathbb{V}|}]|) = O(|\mathbb{E}|)$. In addition to the cost of clustering, each structural similarity computation (*e.g.* $\sigma(u, v)$) takes at least $O(min(|\mathbb{N}[u]|, |\mathbb{N}[v]|))$ time since the computation of structural similarity defined in Definition 14 enumerates all common nodes between $\mathbb{N}[u]$ and $\mathbb{N}[v]$. Therefore, the total running cost of SCAN is $O(\min(|\mathbb{N}[u]|, |\mathbb{N}[v]|)|\mathbb{E}|)$. The average and the largest size of degree are $|\mathbb{E}|/|\mathbb{V}|$ and $|\mathbb{V}|$, respectively. Hence, the average and the worst running cost of SCAN are given by $O(|\mathbb{E}|^2/|\mathbb{V}|)$ and $O(|\mathbb{V}|^3)$, respectively.

5.3 Proposed method: SCAN++

Our goal is to find exactly the same clusters, hubs, and outliers as SCAN from large-scale graphs within short computation time. In this section, we present details of our proposal, SCAN++. We first overview the ideas underlying SCAN++ and then give a full description of the graph clustering algorithm.

5.3.1 Overview of SCAN++

As described in the previous section, SCAN incurs high computation cost for clustering since it entails exhaustive structural similarity computations. In order to efficiently find exactly same clusters as SCAN, we use an observation of real-world graphs: *if node u is two hops away from node v, their structural neighborhoods, $\mathbb{N}[u]$ and $\mathbb{N}[v]$, are likely to share large portion of nodes*. This observation is based on a well-known property of real-world graphs: *real-world graphs are expected to have high clustering coefficients* [3]. For nodes that have high clustering coefficients, the topology among a node and its neighboring nodes is likely to be a clique [108]. Thus, nodes u and v are expected to share most of their neighborhoods if they are two hops apart. For example, in a social network, if a user and friends of his/her friends are in the same community, they are likely to share a lot of common friends even if they do not have direct friendships with each other.

In order to reduce the computation costs, SCAN++ uses a new data structure based on the observation, called *directly two-hop-away reachable node set* (*DTAR* for short), instead of the directly structure-reachable nodes of the original algorithm. Intuitively, DTAR is a set of nodes such that (1) it includes two-hop-away nodes from a given node, and (2) the nodes in DTAR are likely to be lie in the same cluster of the given node. By selecting two-hop-away nodes from the given node, we share the computation of clustering among the given nodes and nodes in DTAR. By using DTAR instead of the structure-reachable nodes of SCAN, we consider two approaches to defeating the exhaustive computation of SCAN. First approach is the *two-phase clustering*. In this method, we reduce the number of similarity computations for clustering without sacrificing the quality of clusters. Specifically, the method first roughly detects subsets of clusters by computing structural similarity only for the pairs of the pivot in DTAR and its adjacent node. It then refines the subsets of clusters to find exactly the same clusters as SCAN. The exactness of clustering results is proved in Section 5.4.2. The second approach is the *similarity*

sharing. In this method, we reduce the computation cost for each similarity computation from $O(|\mathbb{E}|/|\mathbb{V}|)$ by sharing the scores of each structural similarity computation between two-hop-away node pairs in DTAR. We give a detailed definition of DTAR in Section 5.3.2. Also, we discuss the details of the two-phase clustering method and similarity sharing method in Section 5.3.3 and 5.3.4, respectively.

These approaches have two major advantages. The first is that we can extract clusters with small computation cost from large-scale real-world graphs. As described above, our ideas successfully utilize the property of real-world graphs by using DTAR. SCAN++ thus has much lower computation cost for large-scale graphs than SCAN. We discuss the efficiency of SCAN++ in Section 5.4.1. The second advantage is that SCAN++ guarantees the exactness of the clustering results. That is, SCAN++ always provides exactly the same clusters as SCAN. This is because the two-phase clustering method of SCAN++ finds all cores in the given graph. We provide a detailed theoretical analysis of the exactness in Section 5.4.2.

5.3.2 Directly Two-hop-away Reachable (DTAR)

From the observation in the previous section, we first introduce the data structure called DTAR. Intuitively, DTAR is a set of nodes that (1) it includes two-hop-away nodes from a given node, and (2) the nodes in DTAR are likely to lie in the same cluster as the given node. The formal definition of DTAR is as follows:

Definition 19 (Directly two-hop-away reachable) *The definition of DTAR of node u, denoted by $\mathbb{T}[u]$, is given by following equation.*

$$\mathbb{T}[u] = \{v \in \mathbb{V} : v \notin \mathbb{N}_\epsilon[u] \text{ and } \mathbb{N}_\epsilon[u] \cap \mathbb{N}[v] \neq \emptyset\}. \tag{5.7}$$

As a matter of convenience, we call the given node, which acts as the starting point of DTAR, as *pivot* (*i.e.* node u in Definition 19); also, the ϵ-neighborhoods of the pivot are referred as *bridges*.

Similar to the directly structure-reachable nodes given in Definition 16, DTAR is recursively expanded by selecting a new pivot. Specifically, let nodes u and $\mathbb{T}[u]$ be a pivot and a DTAR of node u, respectively; SCAN++ selects node $v \in \mathbb{T}[u]$ as a new pivot and then assigns all nodes in $\mathbb{T}[v]$ to a new DTAR expanded from $\mathbb{T}[u]$. This DTAR, $\mathbb{T}[v]$, expanded from a new pivot in $\mathbb{T}[u]$, is called the *two-hop-away reachable node set* (*TAR* for short). Our proposal recursively finds new

pivots and expands DTARs from the pivots until there are no undiscovered pivots in TAR. After the expansions terminate, SCAN++ obtains a converged TAR rooted at a given node. Formally, the converged TAR, which is rooted at node u, is defined as follows:

Definition 20 (Converged TAR) *Let node u be a pivot. The following equation gives the converged TAR rooted at pivot u, denoted by $\mathbb{T}_u$:*

$$\mathbb{T}_u = \{w \in \mathbb{T}[v] : v \in \mathbb{T}_u\}, \tag{5.8}$$

where $\mathbb{T}_u$ is initially set to $\mathbb{T}_u = \{u\}$.

SCAN++ efficiently detects clusters, which are exactly same as SCAN, from given graphs by using the converged TAR.

5.3.3 Two-phase Clustering

SCAN++ detects the clusters in the given graph by constructing converged TARs and running the two-phase clustering method simultaneously. The two-phase clustering method allows us to efficiently find clusters while matching the exactness of the SCAN results. In this section, we formally introduce this two-phase clustering method.

We overview the two-phase clustering below. The two-phase clustering consists of (1) *local clustering phase* and (2) *cluster refinement phase*. In the local clustering phase, SCAN++ roughly clusters the given graph, and identifies local clusters for each converged TAR. In our algorithm, local clusters are obtained from a converged TAR. The local clusters act as a subset of clusters that are potentially included in the converged TAR. After finding the local clusters, SCAN++ obtains clusters by merging the local clusters in the cluster refinement phase. This refinement phase enables SCAN++ to produce exactly same clustering results as SCAN but with much shorter computation time. We detail each phase does in the following sections.

Local clustering phase

At the beginning of clustering, SCAN++ finds a converged TAR (defined in Definition 20) and then extracts local clusters from the converged TAR, in the bottom-up

clustering manner. By finding local clusters for each converged TAR, SCAN++ captures the rough cluster structures of the given graph. The formal definition of the local cluster is given as follows:

Definition 21 (Local cluster) *If node u is a member of a converged TAR, the following equation gives the local cluster of node u, denoted by $\mathbb{L}[u]$.*

$$\mathbb{L}[u] = \begin{cases} \mathbb{N}_\epsilon[u] & (|\mathbb{N}_\epsilon[u]| \geq \mu) \\ \{u\} & (|\mathbb{N}_\epsilon[u]| < \mu) \end{cases} \tag{5.9}$$

Note that each local cluster in a converged TAR is connected to the other local clusters via bridges. The goal of the local clustering phase is to enumerate all local clusters for each pivot contained in a converged TAR.

Concrete details of the procedure of the local clustering phase are as follows: First, SCAN++ selects arbitrary node $u \in \mathbb{V}$ as a pivot of a DTAR. Next, SCAN++ evaluates the structural similarity defined in Definition 14 for the pivot and its adjacent node that are included in $\mathbb{N}[u]$. By applying Definition 15, SCAN++ then checks whether node u satisfies the requirement of core or not; if $|\mathbb{N}_\epsilon[u]| \geq \mu$, then node u is core. Thus SCAN++ assigns all nodes in $\mathbb{N}_\epsilon[u]$ to $\mathbb{L}[u]$ to the local cluster of node u by applying Definition 21. Otherwise, it only assigns node u to $\mathbb{L}[u]$. After that, SCAN++ obtains the DTAR rooted from node u and selects a new pivot from $\mathbb{T}[u]$. SCAN++ recursively continues this procedure until it finds converged TAR $\mathbb{T}_u$ that is rooted at node u. After termination of the above procedure, SCAN++ selects a new pivot, a node that has not been a pivot or bridge in any converged TAR. SCAN++ terminates the local clustering phase if all nodes are assigned as pivots or bridges.

Efficiency of the local clustering phase: SCAN++ evaluates only the structural similarities between pivots and neighbor bridges; that is, it does not evaluate the structural similarities for adjacent node pairs that are lying between bridges. In the local clustering phase, SCAN++ uses TARs to effectively handle the high clustering coefficient characteristics of real-world graphs. Specifically, let c be the clustering coefficient of a node pair defined by Latapy *et al.* [109]. If node u and v are pivots such that $v \in \mathbb{T}[u]$, the local clustering phase does not compute the structural similarities for $c|\{\mathbb{N}[u] \cup \mathbb{N}[v]\} \backslash \{u, v\}|$ bridges that are shared between node u and v. Hence, as shown in the observation in Section 5.3.1, a high clustering coefficient score c implies that each pivot shares large portion of its structure neighborhoods with the nodes that are two hop-away from the pivot (*i.e.* $c|\{\mathbb{N}[u] \cup \mathbb{N}[v]\} \backslash \{u, v\}|$ bridges). Thus the local clustering phase successfully

prunes the candidates subjected to structural similarity computations for adjacent nodes between other bridges. Theoretical analyses of the efficiency of SCAN++ are shown in Section 5.4.1.

Cluster refinement phase

After identifying the local clusters, SCAN++ then refines them to find exactly the same clusters as SCAN. From Definition 17, we introduce a necessary and sufficient condition for merging local clusters in the following lemma:

Lemma 7 (Merging local clusters) *Let nodes u and v lie in the same converged TAR. We have,*

$$\exists w \in \mathbb{N}_\epsilon[u] \cap \mathbb{N}_\epsilon[v] \textit{ s.t. } |\mathbb{N}_\epsilon[w]| \geq \mu \Leftrightarrow \mathbb{L}[u] \cup \mathbb{L}[v] \subseteq \mathbb{C}[w]. \tag{5.10}$$

Proof We first prove the necessary condition of Lemma 7. Since $w \in \mathbb{N}_\epsilon[u] \cap \mathbb{N}_\epsilon[v]$ *s.t.* $|\mathbb{N}_\epsilon[w]| \geq \mu$, node w is core and we have $u, v \in \mathbb{D}[w]$. From Definition 16 and 21, $\mathbb{L}[u] = \mathbb{N}_\epsilon[u] = \mathbb{D}[u]$ and $\mathbb{L}[v] = \mathbb{N}_\epsilon[v] = \mathbb{D}[v]$, when node u and v are core. Otherwise, $\mathbb{L}[u]$ and $\mathbb{L}[v]$ contain only node u and node v, respectively. Thus we have $\mathbb{L}[u] \cup \mathbb{L}[v] \subseteq \mathbb{D}[w] \cup \mathbb{D}[u] \cup \mathbb{D}[v]$. From Definition 17, we have $\mathbb{C}[w] = \{w \in \mathbb{D}[v] : v \in \mathbb{C}[w]\}$ where $\mathbb{C}[w]$ is initially set to $\mathbb{C}[w] = \{w\}$. Hence, $\mathbb{L}[u] \cup \mathbb{L}[v] \subseteq \mathbb{D}[w] \cup \mathbb{D}[u] \cup \mathbb{D}[v] \subseteq \mathbb{C}[w]$. Therefore, we have the necessary condition of Lemma 7.

Next, we prove the sufficient condition of Lemma 7. Since $\mathbb{L}[u] \cup \mathbb{L}[v] \subseteq \mathbb{C}[w]$, we have $c_u = c_v$. Hence, from Definition 17, we have node w such that $u, v \in \mathbb{C}[w]$ and $|\mathbb{N}_\epsilon[w]| \geq \mu$. Additionally, from Definition 21, nodes u and v are pivots. Recall Definitions 19 and 20, two pivots (*i.e.* node u and v) only share the nodes in $\mathbb{N}_\epsilon[u] \cap \mathbb{N}[v]$ (or $\mathbb{N}[u] \cap \mathbb{N}_\epsilon[v]$). Therefore, node w must be in $\mathbb{N}_\epsilon[u] \cap \mathbb{N}[v]$ (or $\mathbb{N}[u] \cap \mathbb{N}_\epsilon[v]$). Since $u, v \in \mathbb{C}[w]$, nodes u and v have $\sigma(u, w) \geq \epsilon$ and $\sigma(v, w) \geq \epsilon$, respectively. Thus $w \in \mathbb{N}_\epsilon[u] \cap \mathbb{N}_\epsilon[v]$, which yields the sufficient condition of Lemma 7. □

From Lemma 7, if we have core in $\mathbb{N}_\epsilon[u] \cap \mathbb{N}_\epsilon[v]$, $\mathbb{L}[u]$ and $\mathbb{L}[v]$ are assigned to the same cluster. From Definition 21, a local cluster is adjacent to other local clusters via bridges. Hence, if a bridge satisfies the core condition in Definition 15, SCAN++ merges the local clusters adjacent to the bridge into the same cluster.

Intuitively, to find local clusters that are merged into the same cluster, we check all bridges to determine whether they can be cores or not. This is because Lemma 7 implies that we may be able to merge local clusters if a bridge has more than two pivots in its ϵ-neighborhoods. However, this straightforward approach incurs high computation costs since we have to compute structural similarities among cores and bridges. To avoid this inefficient cluster refinement, SCAN++ reuses the results of the local clustering phase. We first define a set of pivots that are included in ϵ-neighborhood of a bridge.

Definition 22 (ϵ-neighborhood pivots of bridge) *Let node b be a bridge extracted during the local clustering phase, the ϵ-neighborhood pivots of node b, denoted by $\mathbb{P}_\epsilon[b]$, are defined as follows:*

$$\mathbb{P}_\epsilon[b] = \{p \in \mathbb{N}[b] : \sigma(b,p) \geq \epsilon \textit{ and } p \textit{ is a pivot}\}. \tag{5.11}$$

From Lemma 7, we have to extract cores from bridges such that $|\mathbb{P}_\epsilon[b]| \geq 2$ since such bridges connects two or more pivots (and their local clusters) with the structural similarity greater than ϵ. However, if the ϵ-neighborhood pivots of a bridge already satisfy the core condition in Definition 15 (*i.e.* $|\mathbb{P}_\epsilon[b]| \geq \mu$ for bridge b) by the local clustering phase, we can determine that the bridge is core without computing the structural similarities. In addition, from Lemma 7 and Definition 22, we can introduce prunable bridges given by the following lemma.

Lemma 8 (Prunable bridges) *Let bridge b be core, and $\bigcup_{p\in\mathbb{P}_\epsilon[b]} \mathbb{L}[p]$ be the merged cluster by Lemma 7. The following set shows prunable bridges that are merged into clusters without computing structural similarities in the subsequent cluster refinement process:*

$$\{b' \in \bigcup_{p\in\mathbb{P}_\epsilon[b]} \mathbb{L}[p] : |\{p' \in \mathbb{P}_\epsilon[b'] : c_{p'} \neq c_b\}| = 0\}, \tag{5.12}$$

where $c_{p'}$ and c_b are clusters of pivot p' and bridge b, respectively.

Proof From Definition 22, prunable bridges have neighborhood pivots whose cluster ids are the same as c_b. This implies that all neighboring local clusters have already been merged in the same cluster by Lemma 7. Hence, the prunable bridges do not merge any local clusters in the subsequent cluster refinement process. □
Lemma 8 implies that we can skip the process to determine the prunable bridges are core nodes at the cluster refinement process.

By using Lemma 7, 8 and Definition 22, we introduce a concrete procedure for the cluster refinement phase as follows: First, SCAN++ obtains a set of bridges $\mathbb{B}$ as a result of the local clustering phase. Next, it selects bridge $b \in \mathbb{B}$ that maximizes $|\mathbb{P}_\epsilon[b]|$ so that we can merge a lot of local clusters and remove many prunable bridges from $\mathbb{B}$ by Lemma 8 if bridge b is core. Then, it determines whether bridge b is core or not. If bridge b is core, SCAN++ merges all nodes in $\bigcup_{p \in \mathbb{P}_\epsilon[b]} \mathbb{L}[p]$ into the same cluster based on Lemma 7. After merging the local clusters, SCAN++ obtains all prunable bridges included in $\{b' \in \bigcup_{p \in \mathbb{P}_\epsilon[b]} \mathbb{L}[p] : |\{p' \in \mathbb{P}_\epsilon[b'] : c_{p'} \neq c_b\}| = 0\}$ by Lemma 8, and removes them from $\mathbb{B}$. These processes are continued until there are no bridges that have more than μ local clusters.

After the above procedure, we can divide the remaining bridges into two groups by their degree: (1) bridges with $|\mathbb{N}[b]| < \mu$, or (2) bridges with $|\mathbb{N}[b]| \geq \mu$ and $2 \leq |\mathbb{P}_\epsilon[b]| < \mu$. From Definition 15, the former case trivially has no cores, hence SCAN++ removes them from $\mathbb{B}$. The latter case may have some cores, so SCAN++ computes the structural similarities only for the bridges in the latter case. Finally, SCAN++ terminates the cluster refinement when there are no unevaluated bridges in $\mathbb{B}$.

Efficiency of the cluster refinement phase: Our cluster refinement phase has short computation time for two reasons: First is that SCAN++ does not require exhaustive structural similarity computations for all bridges. In practice, two local clusters in a converged TAR tend to share a lot of bridges due to the high clustering coefficients of real-world graphs. This implies that we can merge several local clusters at the same time by checking only one of the bridges, and thus prune a lot of computations for prunable bridges included in the merged local clusters (Lemma 8). Therefore, we can reduce the computation time by merging local clusters. Second reason is that structural similarity computations are not required for bridges if the parameter settings are effective. This is based on the observations on the effective parameters (*i.e.* ϵ and μ) for real-world graphs as revealed by Xu *et al.* [41] and Lim *et al.* [104]. In the literature [41], they revealed the following effective parameter setting, given the goal of reasonable clustering results for real-world graphs: *"an ϵ value between 0.5 and 0.8 is normally sufficient to achieve a good clustering result. We recommend a value for μ, of 2."* Also, in the literature [104], Lim *et al.* revealed that clustering quality parameter is less sensitive to μ than ϵ. These observations imply that desirable clustering results can be obtained by properly choosing the above parameters. In practice, if we set parameter $\mu = 2$ based on the observation of the literature [41], the bridges have the following attractive property for efficient computations:

Lemma 9 (Property of bridges for $\mu = 2$) *If we set $\mu = 2$, bridges always satisfy the core condition.*

Proof From the definition of DTAR in Definition 19, SCAN++ always selects bridges from ϵ-neighborhoods of a pivot. In addition, from the definitions of the structural similarity in Definition 14, each node always has the structural similarity that is equal to 1 with itself (*e.g.* $\sigma(u, u) = 1$). As a result, bridges have $|\mathbb{P}_\epsilon[b]| \geq 2$, therefore they always satisfy the core condition when $\mu = 2$. □
That is, bridges in real-world graphs are cores and so structural similarities do not need to be calculated for bridges.

As a result, SCAN++ lowers the computation cost by cluster refinement. We will show that cluster refinement has small, practical computation time for real-world graphs in Section 5.5.2.

5.3.4 Similarity Sharing

In this section, we describe our approach to reducing the cost of structural similarity computation. As shown in Section 5.2, the original algorithm enumerates all common nodes in the structural neighborhoods of two adjacent nodes. This approach is expensive since its time complexity is $O(|\mathbb{E}|/|\mathbb{V}|)$ on average. Therefore, we introduce an efficient method for computing the structural similarity by sharing the intermediate results of structural similarities in DTAR. We first introduce a topological property of DTAR, and then we detail our approach based on the property.

In order to introduce the property, we first define pivot subgraph $\mathbb{G}_w$ by using $\mathbb{T}[u]$ as follows:

Definition 23 (Pivot Subgraph) *If node v is a two-hop-away node from node u (i.e. $v \in \mathbb{T}[u]$) given in Definition 19 and $\mathbb{G}_w = \{\mathbb{V}_w, \mathbb{E}_w\}$ is the pivot subgraph of node w where $\mathbb{V}_w \subseteq \mathbb{V}$ and $\mathbb{E}_w \subseteq \mathbb{E}$, $\mathbb{V}_w$ and $\mathbb{E}_w$ are defined as follows:*

$$\mathbb{V}_w = \mathbb{N}[u] \cap \mathbb{N}[v] \cup \{w\} \tag{5.13}$$

$$\mathbb{E}_w = \{(x, y) \in \mathbb{E} : x, y \in \mathbb{V}_w\} \tag{5.14}$$

Definition 23 indicates that if node v is included in $\mathbb{T}[u]$, we have two pivot subgraphs $\mathbb{G}_u$ and $\mathbb{G}_v$ for node u and v, respectively.

Definition 23 provides the following lemma that shows a topological property of DTAR suggested in Definition 19.

Lemma 10 (Subgraph isomorphism of DTAR) *If node v is a directly two-hop away reachable from node u (i.e. $v \in \mathbb{T}[u]$) given in Definition 19, the pivot subgraphs of node u and v (i.e. $\mathbb{G}_u$ and $\mathbb{G}_v$) are always isomorphic [129].*

Proof From Definition 13 and 19, $\mathbb{N}[u] \cap \mathbb{N}[v] = \{w \in \mathbb{V} : (u, w) \in \mathbb{E} \wedge (v, w) \in \mathbb{E}\} \neq \emptyset$ if node u and v are two-hop-away nodes. Hence, if mapping $\varphi(u) = v$ and $\varphi(w) = w$ where $w \in \mathbb{N}[u] \cap \mathbb{N}[v]$, trivially we have isomorphism mapping $\varphi : \mathbb{V}_u \rightarrow \mathbb{V}_v$ with $(x, y) \in \mathbb{E}_u \Leftrightarrow (\varphi(x), \varphi(y)) \in \mathbb{E}_v$. Therefore, $\mathbb{G}_u$ and $\mathbb{G}_v$ are isomorphic. □

This lemma implies that if node u is a pivot and node v is a node in $\mathbb{T}[u]$ given by Definition 19, node v and the nodes in $\mathbb{N}[u] \cap \mathbb{N}[v]$ always have the same subgraph topology as the subgraph of node u and nodes in $\mathbb{N}[u] \cap \mathbb{N}[v]$. By using Lemma 10, we introduce the following lemma for efficient structural similarity computation.

Lemma 11 (Similarity sharing) *If we have nodes u, v and w such that $v \in \mathbb{T}[u]$ and $w \in \mathbb{N}[u] \cap \mathbb{N}[v]$, we can compute structural similarity $\sigma(v, w)$ by using the result of the structural similarity $\sigma(u, w)$ as follows:*

$$\sigma(v, w) = \frac{\sqrt{|\mathbb{N}[u]||\mathbb{N}[w]|}\sigma(u, w) - |(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]| + |(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|}{\sqrt{|\mathbb{N}[v]||\mathbb{N}[w]|}}. \quad (5.15)$$

Proof From Definition 23, we have two pivot subgraphs $\mathbb{G}_u$ and $\mathbb{G}_v$ for node u and v, respectively. From Lemma 10, subgraphs $\mathbb{G}_u$ and $\mathbb{G}_v$ are isomorphic. Therefore, $\mathbb{N}[u] \cap \mathbb{N}[w]$ shares $\mathbb{N}[u] \cap \mathbb{N}[v] \cap \mathbb{N}[w] \neq \emptyset$ with $\mathbb{N}[v] \cap \mathbb{N}[w]$ since $\mathbb{N}[u] \cap \mathbb{N}[v] \neq \emptyset$ and $w \in \mathbb{N}[u] \cap \mathbb{N}[v]$ for $v \in \mathbb{T}[u]$ given by Definition 19. Hence, if we decompose $|\mathbb{N}[u] \cap \mathbb{N}[w]|$ and $|\mathbb{N}[v] \cap \mathbb{N}[w]|$ by using $\mathbb{N}[u] \cap \mathbb{N}[v] \cap \mathbb{N}[w]$ into $|\mathbb{N}[v] \cap \mathbb{N}[w]| = |\mathbb{N}[u] \cap \mathbb{N}[v] \cap \mathbb{N}[w]| + |(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|$ and $|\mathbb{N}[u] \cap \mathbb{N}[w]| = |\mathbb{N}[u] \cap \mathbb{N}[v] \cap \mathbb{N}[w]| + |(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]|$, we have,

$$\begin{aligned} &|\mathbb{N}[v] \cap \mathbb{N}[w]| = \\ &\quad |\mathbb{N}[u] \cap \mathbb{N}[w]| - |(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]| + |(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|. \end{aligned} \quad (5.16)$$

From Definition 14, structural similarity is as follows:

$$\sigma(v, w) = \frac{|\mathbb{N}[v] \cap \mathbb{N}[w]|}{\sqrt{|\mathbb{N}[v]||\mathbb{N}[w]|}}, \quad (5.17)$$

$$\sigma(u,w) = \frac{|\mathbb{N}[u] \cap \mathbb{N}[w]|}{\sqrt{|\mathbb{N}[v]||\mathbb{N}[w]|}}. \tag{5.18}$$

Hence, from Eq. (5.16) and (5.18),

$$\begin{aligned}\text{Eq. } (5.17) &= \frac{|\mathbb{N}[u] \cap \mathbb{N}[w]| - |(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]| + |(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|}{\sqrt{|\mathbb{N}[v]||\mathbb{N}[w]|}} \\ &= \frac{\sqrt{|\mathbb{N}[u]||\mathbb{N}[w]|}\sigma(u,w) - |(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]| + |(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|}{\sqrt{|\mathbb{N}[v]||\mathbb{N}[w]|}}\end{aligned} \tag{5.19}$$

Therefore, we have Lemma 11. □

Lemma 11 implies that we can reuse the result of the similarity computation $\sigma(u,w)$ for obtaining $\sigma(v,w)$ where node v is a two-hop-away node from node u (*i.e.* $v \in \mathbb{T}[u]$) and $w \in \mathbb{N}[u] \cap \mathbb{N}[v]$.

Efficiency of similarity sharing method: As shown in Lemma 11, SCAN++ shares the scores of structural similarity computations between a node and a node in the DTAR. Hence, SCAN++ reduces the cost of structural similarity computation. From Lemma 10 and 11, the efficiency of the similarity sharing method is as follows:

Lemma 12 (Efficiency of similarity sharing) *Let $v \in \mathbb{T}[u]$ and $w \in \mathbb{N}[u] \cap \mathbb{N}[v]$. The cost for computing the structural similarity $\sigma(v,w)$ is $O(min(|\mathbb{N}[v] \backslash \mathbb{N}[u]|, |\mathbb{N}[w]|))$ if $\sigma(u,w)$ has already been computed.*

Proof From Lemma 11, we can obtain the score of $\sigma(v,w)$ by computing $\sigma(u,v)$, $|(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]|$ and $|(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|$. Given $v \in \mathbb{T}[u]$, we have already have the score of $\sigma(u,w)$ by Definition 19. Additionally, since $(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w] \subseteq \mathbb{N}[u] \cap \mathbb{N}[w]$, $|(\mathbb{N}[u] \backslash \mathbb{N}[v]) \cap \mathbb{N}[w]|$ was also obtained when SCAN++ computed $|\mathbb{N}[u] \cap \mathbb{N}[w]|$ for $\sigma(u,w)$. The remaining term of Eq. (5.15) is just $|(\mathbb{N}[v] \backslash \mathbb{N}[u]) \cap \mathbb{N}[w]|$. Therefore the similarity sharing requires the computational cost $O(min(|\mathbb{N}[v] \backslash \mathbb{N}[u]|, |\mathbb{N}[w]|))$. □

As shown in Section 5.2, the original computation form of the structural similarity in Definition 14 incurs $O(min(|\mathbb{N}[v]|, |\mathbb{N}[w]|)) = O(|\mathbb{E}|/|\mathbb{V}|)$ computation time for average. In contrast, Lemma 12 shows similarity sharing incurs $O(min(|\mathbb{N}[v] \backslash \mathbb{N}[u]|, |\mathbb{N}[w]|))$ when $\sigma(u,w)$ has already been computed. Since the average degree is $|\mathbb{E}|/|\mathbb{V}|$, we have $|\mathbb{N}[v] \backslash \mathbb{N}[u]| \leq |\mathbb{N}[w]| = |\mathbb{N}[v]|$. As a result, $O(min(|\mathbb{N}[v] \backslash \mathbb{N}[u]|, |\mathbb{N}[w]|)) = O(|\mathbb{N}[v] \backslash \mathbb{N}[u]|) \leq O(|\mathbb{E}|/|\mathbb{V}|)$. Therefore similarity sharing computes $\sigma(v,w)$ with smaller computational cost than the original computation form defined by Definition 14.

5.3.5 Algorithm of SCAN++

We can efficiently extract the clustering results by using two-phase clustering and similarity sharing. The pseudo-code of our proposal, SCAN++, is given in Algorithm 4. Algorithm 4 consists of three parts: local clustering phase given by Section 5.3.3 (line 2-17), cluster refinement phase given by Section 5.3.3 (line 18-37), and classification of hubs and outliers (line 38-44). Initially all the nodes are labeled with their own cluster-id (*i.e.* c_u for node u). First, SCAN++ runs local clustering phase (line 2-17). It selects a node as a pivot of a DTAR (line 3-6). Then, SCAN++ computes the structural similarities for the pivot by using Lemma 11 (line 7-9). After that, it finds local clusters from the pivot by Definition 21 (line 10-11). Finally, it expands $\mathbb{T}_u$ by Definition 19 (line 12-13), and continues this procedure until there are no unvisited pivots in $\mathbb{T}_u$. Then, the cluster refinement phase starts. SCAN++ refines local clusters (line 18-37). First, SCAN++ selects bridge b that maximizes $|\mathbb{P}_\epsilon[b]|$ (line 19). If $|\mathbb{N}[b]| < \mu$, the bridge can not be core, and hence it is removed from $\mathbb{B}$ (line 20-21). Otherwise, when $2 \leq |\mathbb{P}_\epsilon[b]| < \mu$, SCAN++ computes the structural similarity of bridge b until SCAN++ can identify node b as core or border (line 23-25). Then, SCAN++ checks if bridge b satisfies the core condition in Definition 15 (line 26). If the bridge is core, SCAN++ merges local clusters by Lemma 7 (line 27-28) and removes prunable bridges from $\mathbb{B}$ based on Lemma 8 (line 29-33). Finally, SCAN++ adds the clusters derived in this phase to $\mathbb{C}$ (line 37). After the cluster refinement, SCAN++ classifies the singleton nodes that do not belong to any cluster, as either hubs or outliers (line 38-44). This phase is based on Definition 18. If a singleton node is adjacent to multiple clusters, it regards the node as a hub (line 38). Otherwise, it regards the node as an outlier (line 40). After assigning all nodes to clusters $\mathbb{C}$, hubs $\mathbb{H}$ or outliers $\mathbb{O}$, SCAN++ terminates the clustering procedure.

Algorithm 4 SCAN++

Input: $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, $\epsilon \in \mathbb{R}$, $\mu \in \mathbb{N}$;
Output: clusters $\mathbb{C}$, hubs $\mathbb{H}$, and outliers $\mathbb{O}$;

$\mathbb{U} = \mathbb{V}$, $\mathbb{B} = \emptyset$, queue Q;
while $\mathbb{U} \neq \emptyset$ **do**
  select a node $u \in \mathbb{U}$;
  $\mathbb{T}_u = \{u\}$;
  while we have unvisited pivots in $\mathbb{T}_u$ **do**
    select node $p \in \mathbb{T}_u$;
    for each node $v \in \mathbb{N}[p]$ **do**
      evaluate $\sigma(p, v)$ by Lemma 11;
    end for
    get $\mathbb{L}[p]$ by Definition 21;
    label all nodes in $\mathbb{L}[p]$ as c_p;
    get $\mathbb{T}[p]$ by Definition 19;
    $\mathbb{T}_u = \mathbb{T}_u \cup \mathbb{T}[p]$;
  end while
  get $\mathbb{V}_{\mathbb{T}_u}$ by Definition 24;
  $\mathbb{U} = \mathbb{U} \backslash \mathbb{V}_{\mathbb{T}_u}$, $\mathbb{B} = \mathbb{B} \cup \{\mathbb{N}_\epsilon[p] \backslash \{p\}\}$;
end while
while $\mathbb{B} \neq \emptyset$ **do**
  get node $b \in \mathbb{B}$ *s.t.* $\arg\max |\mathbb{P}_\epsilon[b]|$;
  if $|\mathbb{N}[b]| < \mu$ **then**
    $\mathbb{B} = \mathbb{B} \backslash \{b\}$;
  else
    if $2 \leq |\mathbb{P}_\epsilon[b]| < \mu$ **then**
      evaluate $\sigma(b, b')$ for $b' \in \mathbb{N}[b] \backslash \mathbb{P}_\epsilon[b]$;
    end if
    if node b is core **then**
      merge $\bigcup_{p \in \mathbb{P}_\epsilon[b]} \mathbb{L}[p]$ in to the same cluster;
      label all nodes in $\bigcup_{p \in \mathbb{P}_\epsilon[b]} \mathbb{L}[p]$ as c_b;
      for each bridge b' in $\bigcup_{p \in \mathbb{P}_\epsilon[b]} \mathbb{L}[p]$ **do**
        if $|\{p \in \mathbb{P}_\epsilon[b'] : c_p \neq c_b\}| = 0$ **then**
          $\mathbb{B} = \mathbb{B} \backslash \{b'\}$ by Lemma 8;
        end if
      end for
    end if
  end if
end while
insert all clusters into $\mathbb{C}$;
for each singleton node $u \in \mathbb{V}$ **do**
  if $\exists x, y \in \mathbb{N}[u]$ *s.t.* $c_x \neq c_y$ **then**
    label node u as *hub* and $u \in \mathbb{H}$;
  else
    label node u as *outlier* and $u \in \mathbb{O}$;
  end if
end for

5.4 Theoretical Analyses of SCAN++

In this section, we theoretically discuss the efficiency and exactness of SCAN++.

5.4.1 Efficiency of SCAN++

We analyze the computational complexity of algorithm SCAN++. Given a graph with $|\mathbb{V}|$ nodes and $|\mathbb{E}|$ edges, SCAN++ finds all clusters *w.r.t.* given parameter settings. This theoretically entails the following time complexity:

Theorem 3 (Time complexity of SCAN++) *SCAN++ incurs time complexity of* $O(\frac{1-c}{c}|\mathbb{E}|)$ *to obtain clustering results where* c *is the average pairwise clustering coefficient [109].*

Proof Let the average degree of the given graph be $|\mathbb{E}|/|\mathbb{V}|$. As we described in Section 5.3.3, each pivot is expected to share $c|\mathbb{E}|/|\mathbb{V}|$ bridges with the other pivots where c is the average pairwise clustering coefficient. Since, from Definition 19 and 20, the shared bridges do not become new pivots, the maximum number of pivots in the given graph is $|\mathbb{V}|^2/(c|\mathbb{E}|)$. Moreover, for each pivot, which acts as a starting point of a DTAR, SCAN++ incurs time complexity of $O(|\mathbb{E}|/|\mathbb{V}|)$. This is because it evaluates structural similarities for all adjacent nodes identified by the pivot. Thus the computational cost for the structural similarity computation for each pivot is $O((|\mathbb{E}|/|\mathbb{V}|)\{|\mathbb{V}|^2/(c|\mathbb{E}|)\}) = O(|\mathbb{V}|/c)$.

In addition, Lemma 12 shows that we can obtain structural similarity on DTARs by computing just $|(\mathbb{N}[v]\backslash\mathbb{N}[u])\cap\mathbb{N}[w]|$ where $v \in \mathbb{T}[u]$ and $w \in \mathbb{N}[u]\cap\mathbb{N}[v]$. Hence, the time complexity of each similarity computation is $O(\min(|\mathbb{N}[v]\backslash\mathbb{N}[u]|, |\mathbb{N}[w]|))$. Recall that a pivot shares $c|\mathbb{E}|/|\mathbb{V}|$ neighborhoods with the other pivots, hence we have $O(\min(|\mathbb{N}[v]\backslash\mathbb{N}[v]|, |\mathbb{N}[w]|)) = O((1-c)|\mathbb{E}|/|\mathbb{V}|)$ time complexity for each structural similarity computation. Therefore, the total time complexity is $O(\frac{|\mathbb{V}|(1-c)|\mathbb{E}|}{c|\mathbb{V}|}) = O(\frac{1-c}{c}|\mathbb{E}|)$. □

As shown in Section 5.5, $|\mathbb{V}| \ll |\mathbb{E}|$ and $0 < c < 1$ in practice. Additionally, as described in Section 5.3.1, it is known that the clustering coefficient c of most real-world graphs tends to be high [3, 108]. Hence, Theorem 3 indicates that the proposed algorithm, SCAN++, can find clustering results much faster than SCAN that needs $O(|\mathbb{E}|^2/|\mathbb{V}|)$.

5.4.2 Exactness of SCAN++

In the previous sections, we introduced a new clustering method SCAN++, which is more efficient than SCAN. However, the following question remains, "Can this approach find exactly the same clusters as SCAN?" We answer this important question affirmatively below.

In order to demonstrate the exactness of the clustering results derived from SCAN++, we show our approach does not fail to find the clusters given by Definition 17. For discussing the exactness of SCAN++, we define a set of nodes that are cluster candidates derived from a converged TAR.

Definition 24 (Candidate clusters) *Let $\mathbb{T}_u$ be a converged TAR obtained by SCAN++, the following equation gives the candidate clusters $\mathbb{V}_{\mathbb{T}_u}$ derived from $\mathbb{T}_u$:*

$$\mathbb{V}_{\mathbb{T}_u} = \mathbb{T}_u \cup \{ \bigcup_{\forall v \in \mathbb{T}_u} \mathbb{N}_\epsilon[v] \}. \tag{5.20}$$

From Definition 24, we have the following lemma:

Lemma 13 (Non-directly structure-reachability) *Let $\overline{\mathbb{V}}_{\mathbb{T}_u}$ be nodes that do not belong to $\mathbb{V}_{\mathbb{T}_u}$ (i.e. $\overline{\mathbb{V}}_{\mathbb{T}_u} = \mathbb{V} \backslash \mathbb{V}_{\mathbb{T}_u}$). SCAN++ always has the following property for adjacent node pair (u, v):*

$$u \in \mathbb{V}_{\mathbb{T}_u} \wedge v \in \overline{\mathbb{V}}_{\mathbb{T}_u} \Rightarrow \sigma(u, v) < \epsilon. \tag{5.21}$$

Proof We prove Lemma 13 by contradiction. We assume that adjacent node pair (v, w) has $\sigma(v, w) \geq \epsilon$ if $v \in \mathbb{V}_{\mathbb{T}_u}$ and $w \in \overline{\mathbb{V}}_{\mathbb{T}_u}$. From Definition 19 and 20, all bridges are adjacent to only the nodes in $\mathbb{V}_{\mathbb{T}_u}$. Thus node v must be a pivot of $\mathbb{T}_u$ since node v is adjacent to node w which belongs to $\overline{\mathbb{V}}_{\mathbb{T}_u}$. Recall Definition 19 that SCAN++ regards ϵ-neighborhoods of a pivot as bridges that are included in $\mathbb{V}_{\mathbb{T}_u}$. Hence, node w is a member of $\mathbb{N}_\epsilon[v] \subseteq \mathbb{V}_{\mathbb{T}_u}$, and this contradicts $w \in \overline{\mathbb{V}}_{\mathbb{T}_u}$. This yields Lemma 13. □

Lemma 13 implies that node set $\mathbb{V}_{\mathbb{T}_u}$ is always surrounded by adjacent nodes whose structural similarities are less than ϵ.

According to Lemma 13, we introduce a property of clusters derived from converged TAR. Informally speaking, the property is that there are no clusters that cross several candidate node sets that originated from different converged TARs. The property is detailed as follows:

Lemma 14 (Cluster comprehensibility) *Let $\mathbb{C}[v]$ be a cluster where node $v \in \mathbb{V}_{\mathbb{T}_u}$. All member nodes included in $\mathbb{C}[v]$ satisfy the following condition:*

$$v \in \mathbb{V}_{\mathbb{T}_u} \Rightarrow \forall w \in \mathbb{C}[v], w \in \mathbb{V}_{\mathbb{T}_u}. \tag{5.22}$$

Proof We prove Lemma 14 by contradiction. At first, we assume the following condition:

$$v \in \mathbb{V}_{\mathbb{T}_u} \Rightarrow \exists w \in \mathbb{C}[v], w \notin \mathbb{V}_{\mathbb{T}_u}. \tag{5.23}$$

From Definition 17, node w is included in the structure-reachable node set of node v since $w \in \mathbb{C}[v]$. However, Lemma 13 shows that node w always has structural similarity less than ϵ for all adjacent nodes in $\mathbb{V}_{\mathbb{T}_u}$ since $w \notin \mathbb{V}_{\mathbb{T}_u}$. Hence, node w is not structure-reachable from node v. This contradicts Eq. (5.23), which yields Lemma 14. □

From Lemma 14, it is clear that there are no clusters that cross several converged TARs; all structure-reachable cores in a candidate cluster $\mathbb{V}_{\mathbb{T}_u}$ always belong to $\mathbb{V}_{\mathbb{T}_u}$. Hence, we can find exactly same clusters as SCAN if we detect all cores that are included in each candidate cluster $\mathbb{V}_{\mathbb{T}_u}$.

Based on Lemma 14, we show that SCAN++ can detect exactly the same clusters as SCAN from each candidate node set as follows:

Theorem 4 (Clustering results of SCAN++) *Our proposed algorithm SCAN++ always has exactly same clustering results as SCAN.*

Proof As shown in Section 5.2, the clusters defined in Definition 17 are uniquely determined by the cores included in a structure-reachable nodes. Additionally, no clusters cross several converged TARs from Lemma 14. Hence, if SCAN++ finds all cores included in the candidate clusters $\mathbb{V}_{\mathbb{T}_u}$ derived from a converged TAR, it clearly produces exactly the same clustering results as SCAN. As shown in Section 5.3.3, SCAN++ finds all cores from pivots since SCAN++ computes the structural similarity for all adjacent nodes of the pivots in the local clustering phase. In addition, SCAN++ can find all cores from the bridges. There are two reasons: (1) as we described in Section 5.3.3, if bridges are adjacent to more than μ pivots with structural similarity that exceeds ϵ, the bridges are trivially regarded as core in the cluster refinement phase, (2) as shown in Algorithm 4 (line 23-25), SCAN++ computes the structural similarity for all remaining bridges, i.e. those whose core condition has not been checked but are adjacent to more than two pivots. Thus, SCAN++ finds all cores in $\mathbb{V}_{\mathbb{T}_u}$. Therefore, the clustering results of SCAN++ are exactly the same as those of SCAN. □

Thus, SCAN++ is assured of yielding exact results as SCAN.

5.5 Experiments

We compared the effectiveness of four algorithms including our proposed method SCAN++.

- **SCAN++**: our proposal *with* similarity sharing.
- **SCAN***: a simple variation of SCAN that produces approximate results by utilizing the edge sampling technique proposed by the state-of-the-art method LinkSCAN* [104]. Based on LinkSCAN*, SCAN* samples $\min\{d_u, \alpha + \beta \ln d_u\}$ edges for each node, where d_u is the degree of a node and both α and β are user-specified parameters. We set $\alpha = 2|\mathbb{E}|/|\mathbb{V}|$ and $\beta = 1$ as recommended by LinkSCAN*.
- **SCAN**: the original algorithm [41].
- **gSkeletonClu**: a state-of-the-art algorithm extended from SCAN that provides us parameter-free structural clustering [101]. gSkeletonClu employs the tree-decomposition-based algorithm and it searches clustering results that maximize the score of modularity [34].

Our experiments will demonstrate that:

- **Efficient**: Compared to existing algorithms, SCAN++ achieves higher speed clustering for large-scale graphs (Section 5.5.2)
- **Exact**: SCAN++ yields exactly the same clustering results as SCAN (Section 5.5.3)
- **Effective**: SCAN++ is effective in improving clustering speed for large-scale graphs whose clustering coefficients are high (Section 5.5.4)
- **Scalable**: SCAN++ has near-linear scalability against graph size in terms of running time (Section 5.5.5)

Our experiments were designed to show that the proposed approach is a very viable option for computing clusters, hubs, and outliers.

All experiments were conducted on a Linux 2.6.18 server with one CPU (Intel Xeon Processor L5640 2.27GHz) and 144GBytes of main memory. SCAN++, SCAN* and SCAN were implemented in C/C++ using the gcc-g++ 4.8.1 compiler and we used the optimization parameter "-O2" for each algorithm. To evaluate the other algorithm, we used the program of gSkeletonClu published on their authors' sites[1].

[1] `http://web.xidian.edu.cn/jbhuang/en/publications.html`

Table 5.2: Real-world datasets

Dataset	$\lvert\mathbb{V}\rvert$	$\lvert\mathbb{E}\rvert$	c
condmat	23,133	186,936	0.6334
slashdot	77,360	905,468	0.0555
amazon	334,863	925,872	0.3967
dblp	317,080	1,049,866	0.6324
road	1,379,917	3,843,320	0.0470
cnr	325,557	5,477,938	0.5586
google	875,713	5,105,039	0.5143
skitter	1,696,415	11,095,298	0.2581

5.5.1 Datasets

The experiments used the following eight public datasets published by Standard Network Analysis Project[2] and Laboratory of Web Algorithmics[3]:

- *condmat*: This is a researcher collaboration network extracted from Arxiv Condense Matter Physics papers; each node represents an author and each edge represents co-authorship between users [130].
- *shlashdot*: This is an online social network in Slashdot. Nodes correspond to users and edges correspond to friend/foe relationship between users [131].
- *amazon*: This is a co-purchasing network at Amazon, where each node and edge represent a product and a relationship between products that are frequently co-purchased, respectively [132].
- *dblp*: This is a researcher collaboration graph extracted from the bibliography service DBLP; each node is an author and each edge represents coauthor relationship [132].
- *road*: This is a road network of Texas, where nodes correspond to intersections and endpoints and the roads connecting these intersections or endpoints are represented by edges [131].

[2] `http://snap.stanford.edu`
[3] `http://law.di.unimi.it/datasets.php`

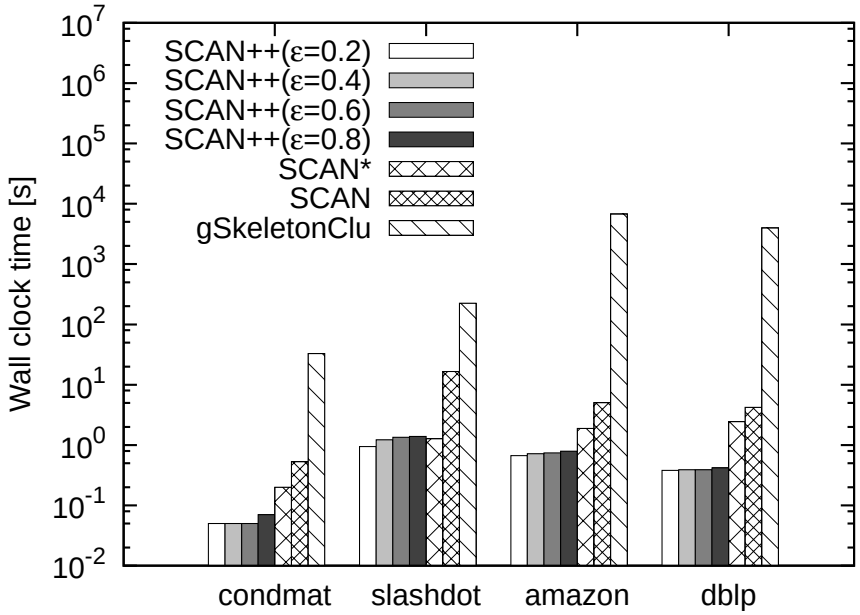

Figure 5.1: Running time for smaller real-world datasets

- *cnr*: This is a web graph crawled from the Italian CNR domain; nodes and edges represent web pages and hyperlinks between two web pages, respectively.
- *google*: This is a web graph released by Google as a part of Google Programming Contest; each node represents a web page and each edge represents a hyperlink between web pages [131].
- *skitter*: This is an internet topology graph obtained by the traceroute command; each node represents a IP address and each edge represents a connection between IP addresses [131].

The statistics of each dataset are shown in Table 5.2. In the right most column, c shows the average clustering coefficient. As a matter of convenience, we refer to the group of datasets condmat, slashdot, amazon and dblp as the smaller datasets, and the remaining datasets, road, cnr, google and skitter, as the larger datasets. Additionally, in order to evaluate the effectiveness of our algorithm, we also used synthetic datasets generated by LFR benchmark [112], which is considered as the *de facto standard* model for generating graphs. The settings will be detailed later.

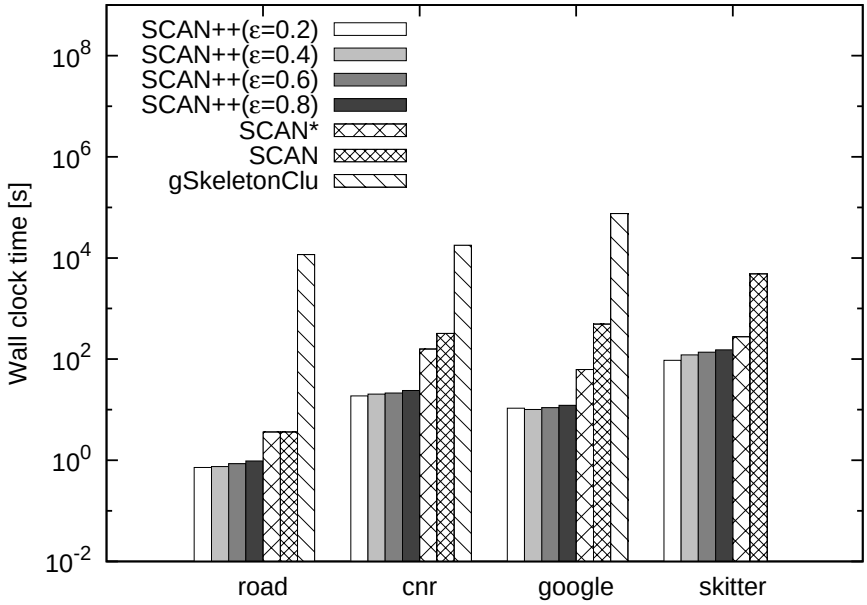

Figure 5.2: Running time for larger real-world datasets

5.5.2 Efficiency

We evaluated the clustering performance of each method through wall clock time for the real-world datasets. In this evaluation, we fixed the parameter $\mu = 5$ and varied the parameter ϵ as 0.2, 0.4, 0.6 and 0.8 for each algorithm. Figure 5.2 shows the running time for each real-world dataset. Since existing algorithm show almost same results under all parameter settings, we omitted the results of them from Figure 5.2 except for $\epsilon = 0.6$. In addition, we omitted the results of gSkeletonClu for skitter since it cannot compute clusters in a day.

Figure 5.2 shows that SCAN++ is much faster than existing approaches under all conditions examined. Of particular interest, SCAN++ is 20 times faster than SCAN on average, and it is also a few orders of magnitude faster than gSkeletonClu. As described in Section 5.2, SCAN subjects all adjacent nodes in the given graph to structural similarity computations. Furthermore, SCAN incurs average computation time of $O(|\mathbb{E}|/|\mathbb{V}|)$ for each structural similarity computation. Hence, SCAN requires $O(|\mathbb{E}|^2/|\mathbb{V}|)$ time on average. Similar to SCAN, for finding clustering results that maximizes modularity, gSkeletonClu has to extract spanning trees from the graph by computing structural similarities for all adjacent nodes. Therefore, as

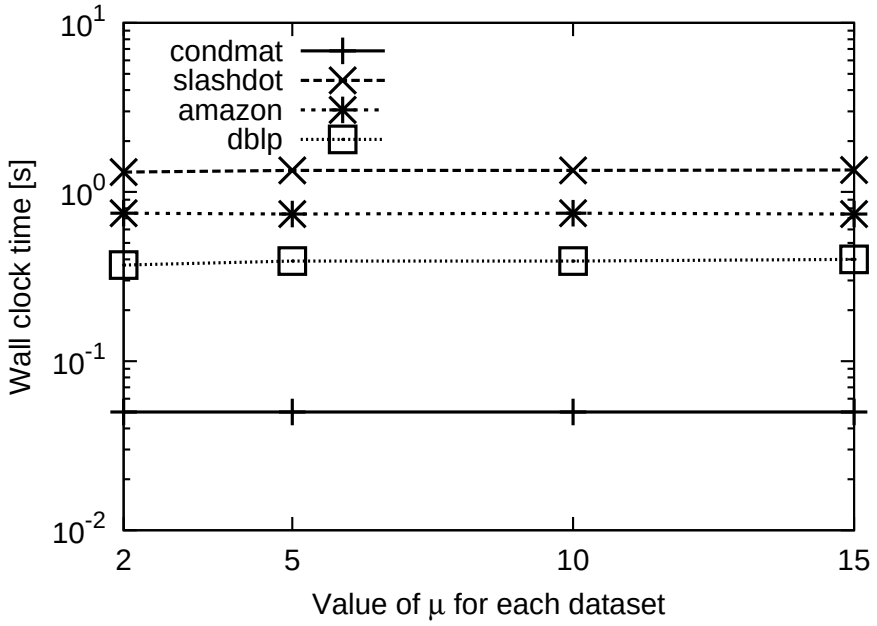

Figure 5.3: Parameter μ differences for smaller datasets ($\epsilon = 0.6$)

shown in Figure 5.2, gSkeletonClu requires significantly larger computation times for clustering than SCAN++ or SCAN. In contrast to both SCAN and gSkeletonClu, as shown in Section 5.3, SCAN++ employs two efficient clustering approaches, (1) two-phase clustering and (2) similarity sharing that utilizes the clustering coefficient. As a result, as shown in Theorem 3, SCAN++ only requires near-linear complexity $O(\frac{1-c}{c}|\mathbb{E}|)$ in terms of number of edges. Therefore, SCAN++ can find clusters, hubs and outliers much more efficiently than SCAN*, SCAN and gSkeletonClu.

Figure 5.2 also shows that SCAN* could be competitive with our proposal SCAN++ for slashdot in terms of efficiency. This is due to former's use of the clustering coefficient of the given graph. As shown in Table 5.2, slashdot has a significantly lower clustering coefficient than the other datasets. SCAN++ could not reduce the running time enough by using two-phase clustering and similarity sharing since the small graphs had low clustering coefficients. Even though road and skitter have relatively lower clustering coefficients than the other datasets, SCAN++ was much faster than SCAN*. There are two reasons. First, road and skitter are much larger graphs than slashdot. If graph size is large enough, SCAN++ can reduce the computation time even if the clustering coefficients are small. Second,

each node in road has almost the same degree while slashdot has a skewed degree distribution. As we described, SCAN* eliminates edges from the graph when the degree of each node is large enough. However, the nodes in road have almost the same degree; hence SCAN* could not effectively eliminate edges from the dataset. Therefore, SCAN++ ran faster than SCAN* for road and skitter. Although, SCAN* is efficient for small graphs with lower clustering coefficients, it is an approximation approach based on SCAN and so can not match the clustering performance of the other methods. We will discuss this point in Section 5.5.3.

In all conditions examined, the running time of the cluster refinement phase described in Section 5.3.3 is negligible. Specifically, SCAN++ consumed less than 1% of its running time for merging clusters under all conditions examined. This is because, in the real-world datasets with high clustering coefficients, each bridge is adjacent to many pivots with high structural similarity scores. Hence, as shown in Lemma 8, most bridges are prunable bridges, and they do not require additional similarity computations for merging local clusters. We omit the detailed results of the running time for merging local clusters due to space limitations.

Our experiments also considered different parameter μ settings. Figure 5.3 shows the running time of SCAN++ for the smaller datasets for various values of μ. As shown in Figure 5.3, the values of μ have no significant impact for the running time of SCAN++. This is because the running time of the cluster refinement phase consumes at most 1% of the total running time. Although we omit the results of the other algorithms from Figure 5.3 and the results of the larger datasets due to space limitations, all the other methods shows almost same results as Figure 5.2. SCAN++ can find clusters, hubs, and outliers more efficiently than the existing approaches even under different parameter settings.

5.5.3 Exactness

One major contribution of SCAN++ is that it outputs exactly same clustering results as SCAN. To demonstrate the exactness of the clustering results, we evaluated accuracy of obtaining cores against SCAN for each dataset. This is because, as shown in Definition 17, clustering results are uniquely determined by the cores. In this experiment, we used F-measure as the metrics of accuracy [133]. F-measure quantifies the accuracy of the clustering results by calculating the harmonic mean of precision and recall. Hence, we defined precision and recall as follows: precision is the fraction of cores by each method that matches those of SCAN, and recall

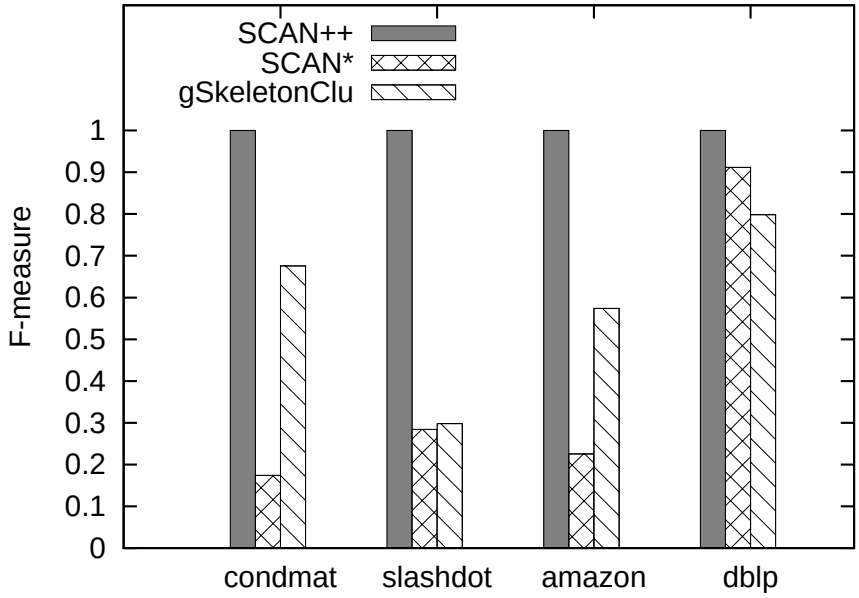

Figure 5.4: F-measure for smaller real-world datasets ($\epsilon = 0.6, \mu = 5$)

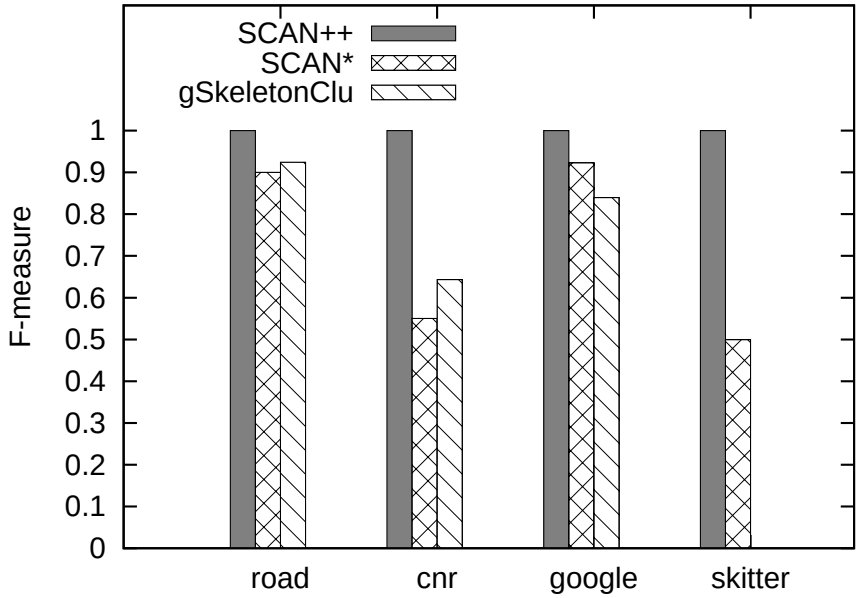

Figure 5.5: F-measure for larger real-world datasets ($\epsilon = 0.6, \mu = 5$)

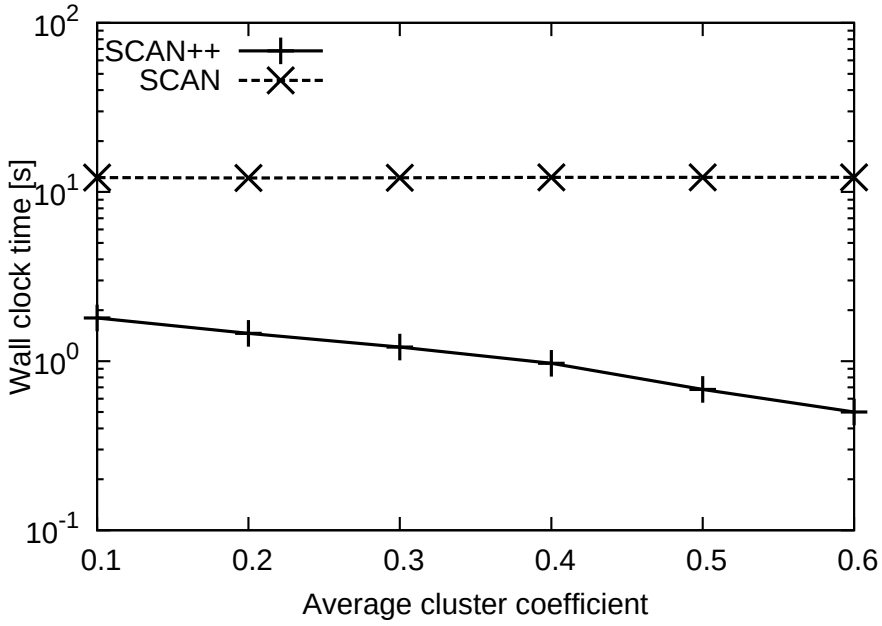

Figure 5.6: c differences for each algorithm

is the fraction of cores obtained by SCAN that are also extracted by each method. F-measure takes a value between 0 and 1, and F-measure is 1 if the obtained cores exactly match those by SCAN. Figure 5.5 shows F-measure of each method against SCAN. In this evaluation, we set the parameters of each algorithm as $\epsilon = 0.6$ and $\mu = 5$. As well as Figure 5.2, we omitted the results of gSkeletonClu for skitter since it does not return the clustering result in a day.

Figure 5.5 indicates that SCAN++ can obtain exactly same clustering results as SCAN. Even though we drops unnecessary similarity computations, SCAN++ guarantees of outputting the same cores as SCAN as shown in Theorem 4. Therefore, F-measure of SCAN++ were 1 as shown in Figure 5.5. On the other hand, SCAN* and gSkeletonClu output clustering results that differ from those of SCAN. This is because SCAN* is an approximation method that samples subset of edges from the given graph and gSkeletonClu employs the clustering results that maximize modularity [34]. Figure 5.5, as well as Figure 5.2, confirms that SCAN++ is superior to existing methods in terms of speed and accuracy.

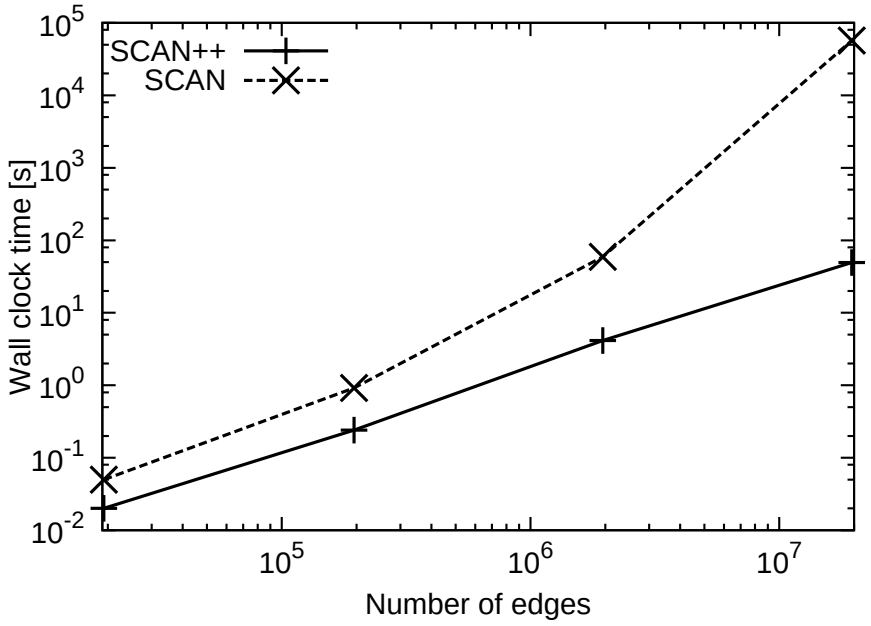

Figure 5.7: Scalability for each algorithm

5.5.4 Effectiveness

We evaluate the effectiveness of our algorithm in terms of high clustering coefficients. To evaluate the effectiveness, we used synthetic graphs produced by the LFR benchmark. We generate LFR benchmark graphs with 1,000,000 nodes; the average clustering coefficient was varied from 0.1 to 0.6 following the real-world datasets in Table 5.2. The other parameters, average degree and maximum degree, were fixed at 20 and 50, respectively. Figure 5.6 shows the computation times of our proposal and SCAN for difference c scores. As shown in Figure 5.6, SCAN shows almost constant computation time under all conditions examined. Unlike SCAN, our algorithm increased its clustering speed as c increased. In the most efficient case (*i.e.* $c = 0.6$) our proposals was up to three times faster than the result of the worst case (*i.e.* $c = 0.1$). These results imply that our two-hop-away node based algorithm effectively prunes the candidates that are assessed. Thus, our algorithm outperforms SCAN when the given graph has high c scores as is likely with real-world graphs.

5.5.5 Scalability

We evaluated the scalability of SCAN++ and SCAN. We generated LFR benchmark graphs with various numbers of nodes from 1,000 to 1,000,000. The other parameters, average degree, maximum degree and clustering coefficient, were fixed at 20, 50 and 0.4, respectively. The running times of the algorithms, shown in Figure 5.7, show that SCAN++ has near-linear scalability in terms of number of edges. On the contrary, SCAN exponentially increases its running time with number of edges. This result verifies our theoretical analysis in Section 5.4.1, hence, our proposals are scalable for large-scale graphs.

5.6 Summary of this Chapter

This chapter addressed the problem of efficiently finding clusters, hubs, and outliers in large-scale graphs. Our proposal, SCAN++, is based on three ideas: (1) it introduces a new data structure, called directly two-hop-away reachable node set (DTAR), that contains only nodes that two hops away from a given node, (2) it drops unnecessary density evaluations for adjacent nodes in the clustering procedure by using DTARs, and (3) its density evaluation method is highly efficient since it shares some of the density evaluation results of DTARs. As a result, SCAN++ has all of the following attractive advantages:

- **Efficient**: SCAN++ achieves 20 times higher clustering speed than its competitors; it scales very well, offering linear clustering time against the number of edges in the graph.
- **Exact**: SCAN++ is not an approximation method. It always returns exactly the same clustering results as SCAN.
- **Effective**: SCAN++ is effective in improving the clustering speed for real-world graphs with high clustering coefficients.

As far as we know, this is the first study to introduce a graph clustering algorithm that achieves both high speed and exact clustering results at the same time. Graph clustering algorithms that extract not only clusters but also hubs and outliers are essential for many applications. The proposal will help to improve the effectiveness of current and future applications.

Chapter 6

Conclusion and Future Work

This dissertation addressed the problem of efficiently finding clusters in large-scale graphs. Our graph clustering proposals use the topological properties of real-world graphs such as clustering coefficient and power-law distribution for efficient clustering. As far as we know, this is the first study to introduce graph clustering algorithms that achieve both high speed and highly accurate clustering results at the same time. The contributions of this dissertation consist of three sub-works: (1) fast algorithm and (2) parallel algorithm for modularity-based graph clustering (Chapters 3 and 4, respectively), and (3) efficient algorithm for density-based graph clustering (Chapter 5). Our experiments on both real-world and synthetic datasets showed that our proposals are much faster than state-of-the-art algorithms for large-scale graphs. Graph clustering algorithms are essential for many applications. These proposals will help to improve the effectiveness of current and future applications.

6.1 Summary of Contributions

We summarize the contributions of this thesis as follows:

- We introduced an efficient modularity-based graph clustering algorithm, named IMAC. Our experimental evaluation reported that IMCA is almost 60 times faster than the state-of-the-art algorithm BGLL. Specifically, it computes clusters from a real-world graph with more than 100 million nodes and 1 billion

edges within 156 seconds (Chapter 3).

- We investigated a vectorized variant of BGLL, named ParBGLL, for modularity clustering. We evaluated our approach on both real-world graphs and synthetic graphs, and showed that our approach is up to 3 times faster than the default setting of BGLL (Chapter 4).

- We proposed an efficient clustering algorithm, named SCAN++, to address structural clustering problems. We proved that SCAN++ offers the same clustering quality as the state-of-the-art algorithm SCAN; SCAN++ always returns exactly same clustering results as SCAN. In addition, our experiments reported that SCAN++ is almost 20 times faster than SCAN for real-world datasets (Chapter 5).

We believe that this work not only enhances the possibility of analyzing large-scale real-world graphs, but also contributes to scientific and/or cultural advances.

6.2 Future Work

We state our future work for each chapter and our long-term view.

Future work based on Chapters 3 and 5 has two parts. The first is to investigate fast and highly accurate clustering algorithms for time evolving graphs. Recently, the topology of real-world graphs changes dynamically and the volume of the changes is significant. For example, the micro-blogging service Twitter reported that up to 340 million daily posts were logged in 2012 [134]. For this reason, we believe that incremental graph clustering is critical. The second is to investigate parameter free graph clustering. As we described in Chapter 5, our proposed method SCAN++ requires some user-specified parameters. However, different datasets are likely to have different parameter settings. Therefore, such parameters should be automatically tuned through an understanding of the statistical properties of real-world graphs.

Future work for Chapter 4 is also twofold. The first is to investigate more scalable data parallel computation techniques by using graph compression. As shown in Chapter 4, the parallelism of our method is highly dependent on SIMD register size. This, however, limits the degree of parallelism possible since the largest SIMD

register is (as of 2014) still less than 512 bits. In order to improve the scalability of the SIMD register, an intuitive approach is to introduce graph compression. The second is to propose task parallel techniques that do not sacrifice clustering quality. In order to introduce task parallel graph clustering, we first partition the input graph into several subgraphs before the clustering procedure. However, this partitioning may degrade the clustering quality since it can split some clusters. In order to achieve highly scalable graph clustering, one of the key challenges to introduce a task parallel method that does not degrade clustering quality.

Finally, we present the long-term goal. In the Big Data era, we must be able to handle the volume and variety of data. Against the problem of data volume, efficient data mining algorithms, which are not limited to just graph mining algorithms, are highly demanded to uncover and understand the real-world data around us. To deal with data variety, we plan to investigate mining algorithms that handle the attributes and semantics of data.

Bibliography

[1] P. Erdos and A Renyi. On the Evolution of Random Graphs. In *Publication of the Mathematical Institute of the Hungaring Academy of Sciences*, volume 5, pages 17–61, 1960.

[2] Stanley Milgram. The Small World Problem. *Psychology Today*, 67(1):61–67, 1967.

[3] D. J. Watts and S. H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393(6684):409–10, 1998.

[4] Albert-Laszlo Barabasi and Reka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[5] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-law Relationships of the Internet Topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, New York, NY, USA, 1999. ACM.

[6] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[7] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast Incremental and Personalized PageRank. pages 173–184, 2010.

[8] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 349–360. ACM, 2013.

[9] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Dynamic and Historical Shortest-Path Distance Queries on Large Evolving Networks by Pruned Landmark Labeling. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, pages 237–248. ACM, 2014.

[10] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-ichi Kawarabayashi. Fast and Accurate Influence Maximization on Large Networks with Pruned Monte-Carlo Simulations. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 138–144. AAAI Press, 2014.

[11] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 75–86. ACM, 2014.

[12] Manish Gupta, Jing Gao, Xifeng Yan, Hasan Cam, and Jiawei Han. Top-k Interesting Subgraph Discovery in Information Networks. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 820–831. IEEE, 2014.

[13] Wenqing Lin, Xiaokui Xiao, and Gabriel Ghinita. Large-scale Frequent Subgraph Mining in MapReduce. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 844–855. IEEE, 2014.

[14] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. Clustering Large Attributed Graphs: an Efficient Incremental Approach. In *ICDE*, pages 689–698, 2010.

[15] Tiancheng Lou and Jie Tang. Mining Structural Hole Spanners Through Information Diffusion in Social Networks. In *WWW*, pages 825–836, 2013.

[16] Mahmoud Taghizadeh and Subir Biswas. Community Based Cooperative Content Caching in Social Wireless Networks. In *Proc. MobiHoc*, pages 257–262, 2013.

[17] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. BioGRID: A General Repository for Interaction Datasets. *Nucleic acids research*, 34(suppl 1):D535–D539, 2006.

[18] Suraj Peri, J Daniel Navarro, Troels Z Kristiansen, Ramars Amanchy, Vineeth Surendranath, Babylakshmi Muthusamy, TKB Gandhi, KN Chandrika, Nandan Deshpande, Shubha Suresh, et al. Human Protein Reference Database as a Discovery Resource for Proteomics. *Nucleic acids research*, 32(suppl 1):D497–D501, 2004.

[19] Andrew Chatr-Aryamontri, Arnaud Ceol, Luisa Montecchi Palazzi, Giuliano Nardelli, Maria Victoria Schneider, Luisa Castagnoli, and Gianni Cesareni. MINT: the Molecular INTeraction Database. *Nucleic acids research*, 35(suppl 1):D572–D574, 2007.

[20] Juan Casado-Vela, Rune Matthiesen, Susana Selles, and Jose Ramon Naranjo. Protein-Protein Interactions: Gene Acronym Redundancies and Current Limitations Precluding Automated Data Integration. *Proteomes*, 1(1):3–24, 2013.

[21] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[22] Zhichao Liu, Qiang Shi, Don Ding, Reagan Kelly, Hong Fang, and Weida Tong. Translating Clinical Findings into Knowledge in Drug Safety Evaluation - Drug Induced Liver Injury Prediction System (DILIps). *PLoS Computational Biology*, 7(12), 2011.

[23] Sarath Chandra C. Janga and Andreas Tzakos. Structure and Organization of Drug-target Networks: Insights from Genomic Approaches for Drug Discovery. *Molecular bioSystems*, 5(12):1536–1548, Dec 2009.

[24] Olaf Sporns, Giulio Tononi, and Rolf Kötter. The Human Connectome: A Structural Description of the Human Brain. *PLoS Computational Biology*, 1(4):e42, 2005.

[25] Di Yu, Tianji Wu, Yi Shan, Yu Wang, Yong He, and Ningyi Yang. Making Human Connectome Faster: CPU Acceleration of Brain Network Analysis. In *Proceedings of IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, pages 593–600. IEEE, 2010.

[26] E. Bullmore and O. Sporns. Complex Brain Networks: Graph Theoretical Analysis of Structural and Functional Systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[27] Yong He, Zhang J. Chen, and Alan C. Evans. Small-world Anatomical Networks in the Human Brain Revealed by Cortical thickness from MRI. *Cerebral Cortex*, 17(10):2407–2419, 2007.

[28] Karl W. Doron, Danielle S. Bassett, and Michael S. Gazzaniga. Dynamic Network Structure of Interhemispheric Coordination, 2012.

[29] Maurice de Kunder. WorldWideWebSize.com || The size of the World Wide Web (The Internet). `http://worldwidewebsize.com/`, 2014.

[30] Inc Facebook. Facebook Reports Third Quarter 2014 Results. `http://investor.fb.com/releasedetail.cfm?ReleaseID=878726`, 2014.

[31] Anja Jentzsch, Richard Cyganiak, and Chris Bizer. State of the LOD Cloud. `http://lod-cloud.net/state/`, 2014.

[32] Réka Albert and Albert-László Barabási. Statistical Mechanics of Complex Networks. *Rev. Mod. Phys.*, 74(1):47–97, Jan 2002.

[33] P. W. Holland and S. Leinhardt. Transitivity in Structural Models of Small Groups.

[34] M. E. J. Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Phys. Rev. E*, 69:026113, Feb 2004.

[35] M. E. J. Newman. Fast Algorithm for Detecting Community Structure in Networks. *Physical Review E - PHYS REV E*, 69:066133, Jun 2004.

[36] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding Community Structure in Very Large Networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[37] Ken Wakita and Toshiyuki Tsurumi. Finding Community Structure in Mega-Scale Social Networks. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 1275–1276. ACM, 5 2007.

[38] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008:P10008, October 2008.

[39] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[40] Santo Fortunato and Marc Barthélemy. Resolution Limit in Community Detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 1 2007.

[41] Xiaowei Xu, Nurcan Yuruk, Zhidan Geng, and Thomas A. J. Schweiger. SCAN: A Structural Clustering Algorithm for Networks. In *KDD*, pages 824–833, 2007.

[42] Vic Grout and Stuart Cunningham. A Constrained Version of a Clustering Algorithm for Switch Placement and Interconnection in Large Networks", booktitle = Proceedings of the 19th International Conference on Computer Applications in Industry and Engineering (CAINE), pages = 252-257, year = 2006,.

[43] Vic Grout, Stuart Cunningham, and Rich Picking. Practical Large-scale Network Design with Variable Costs for Links and Switches. *International Journal of Computer Science and Network Security*, 7(7):113–125, 2007.

[44] Satu Virtanen. Clustering the Chilean Web. In *Proceedings of the 1st Latin American Web Congress (LAWEB)*, pages 229–231. IEEE, 2003.

[45] KilHong Joo and WonSuk Lee. An Incremental Document Clustering for the Large Document Database. In *Information Retrieval Technology*, volume 3689 of *Lecture Notes in Computer Science*, pages 374–387. Springer Berlin Heidelberg, 2005.

[46] Ting-Chao Hou and Tzu-Jane Tsai. A Access-based Clustering Protocol for Multihop Wireless ad hoc Networks. *Selected Areas in Communications, IEEE Journal on*, 19(7):1201–1210, Jul 2001.

[47] C. R. Lin and M. Gerla. Adaptive Clustering for Mobile Wireless Networks. *IEEE Journal on Selected Areas in Communications*, 15(7):1265–1275, Sep 2006.

[48] T.N. Dinh, Ying Xuan, and M.T. Thai. Towards Social-aware Routing in Dynamic Communication Networks. In *Proceedings of 28th IEEE International Conference on Performance Computing and Communications Conference (IPCCC 2009)*, pages 161–168, Dec 2009.

[49] Nam P. Nguyen, Thang N. Dinh, Ying Xuan, and My T. Thai. Adaptive Algorithms for Detecting Community Structure in Dynamic Social Networks. In *Proceedings of the 30th IEEE International Conference on Computer Communications (IEEE INFOCOM 2011)*, pages 2282–2290, 3 2011.

[50] Takehiro Furuta, Mihiro Sasaki, Fumio Ishizaki, Atsuo Suzuki, and Hajime Miyazawa. A New Cluster Formation Method for Sensor Networks using Facility Location Theory. Technical report, Tech. Rep. NANZAN-TR-2006-01, Nanzan Academic Society Mathematical Sciences and Information Engineering, Nagoya, Japan, 2006.

[51] Guan Wang, Yuchen Zhao, Xiaoxiao Shi, and Philip S. Yu. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 588–596, New York, NY, USA, 2012. ACM.

[52] Wenjun Zhou, Hongxia Jin, and Yan Liu. Community Discovery and Profiling with Social Messages. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 388–396, New York, NY, USA, 2012. ACM.

[53] Marek Ciglan, Michal Laclavík, and Kjetil Nørvåg. On Community Detection in Real-world Networks and the Importance of Degree Assortativity. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1007–1015, New York, NY, USA, 2013. ACM.

[54] Yi-Cheng Chen, Wen-Yuan Zhu, Wen-Chih Peng, Wang-Chien Lee, and Suh-Yin Lee. CIM: Community-Based Influence Maximization in Social Networks. *ACM Trans. Intell. Syst. Technol.*, 5(2):25:1–25:31, April 2014.

[55] Pei Lee, Laks V. S. Lakshmanan, and Evangelos E. Milios. Incremental Cluster Evolution Tracking from Highly Dynamic Detwork Data. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE2011)*, pages 3–14, 2014.

[56] Jiwei Li and Claire Cardie. Timeline Generation: Tracking Individuals on Twitter. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 643–652, New York, NY, USA, 2014. ACM.

[57] Hao Ma, Haixuan Yang, Michael R. Lyu, and Irwin King. Mining Social Networks Using Heat Diffusion Processes for Marketing Candidates Selection. In *CIKM*, pages 233–242, 2008.

[58] Kristina Toutanova and Christopher D. Manning. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-speech Tagger. In *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*, EMNLP '00, pages 63–70, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[59] Ying Xu, Victor Olman, and Dong Xu. Clustering Gene Expression Data using a Graph-Theoretic Approach: an Application of Minimum Spanning Trees. *Bioinformatics*, 18(4):536–545, 2002.

[60] Frdric Boyer, Anne Morgat, Laurent Labarre, Jol Pothier, and Alain Viari. Syntons, Metabolons and Interactons: An Exact Graph-Theoretical Approach for Exploring Neighbourhood Between Genomic and Functional Data. *Bioinformatics*, 21(23):4209–4215, 2005.

[61] Yijun Ding, Minjun Chen, Zhichao Liu, Don Ding, Yanbin Ye, Min Zhang, Reagan Kelly, Li Guo, Zhenqiang Su, StephenC Harris, Feng Qian, Weigong Ge, Hong Fang, Xiaowei Xu, and Weida Tong. atBioNet An Integrated Network Analysis Tool for Genomics and Biomarker Discovery. *BMC Genomics*, 13(1), 2012.

[62] Yang Wang, D. Chakrabarti, Chenxi Wang, and C. Faloutsos. Epidemic Spreading in Real Networks: an Eigenvalue Viewpoint. In *SRDS*, pages 25–34, 2003.

[63] Mark EJ Newman. Properties of Highly Clustered Networks. *Physical Review E*, 68(2):026121, 2003.

[64] Ajit A. Diwan, Sanjeeva Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB)*, pages 342–353, 1996.

[65] Andrew Y. Wu, Michael Garland, and Jiawei Han. Mining Scale-free Networks Using Geodesic Clustering. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 719–724, New York, NY, USA, 2004. ACM.

[66] Rakesh Agrawal and H. V. Jagadish. Algorithms for Searching Massive Graphs. *IEEE Trans. Knowl. Data Eng.*, 6(2):225–238, 1994.

[67] Paul S Bradley, Usama M Fayyad, Cory Reina, et al. Scaling Clustering Algorithms to Large Databases. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 9–15, 1998.

[68] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling Liu. SemStore: A Semantic-Preserving Distributed RDF Triple Store. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, pages 509–518. ACM, 2014.

[69] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 5(8):716–727, 2012.

[70] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proc. ACM SIGMOD 2013*, pages 505–516, 2013.

[71] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[72] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 8 1998.

[73] François Pellegrini and Jean Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1996, Brussels, Belgium, April 15-19, 1996, Proceedings*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996.

[74] Tom Chao Zhou, Hao Ma, Michael R. Kyu, and Irwin King. UserRec: A User Recommendation Framework in Social Tagging Systems. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*. AAAI Press, 2010.

[75] B W Kernighan and S Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical journal*, 49(2):291–307, 2 1970.

[76] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An Algorithmic Framework for Performing Collaborative Filtering. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1999)*, pages 230–237. ACM, 1999.

[77] Thomas Hofmann. Latent Semantic Models for Collaborative Filtering. *ACM Transactions on Information Systems (TOIS)*, 22(1):89–115, 2004.

[78] Yiming Yang, Thomas Pierce, and Jaime G. Carbonell. A Study of Retrospective and On-Line Event Detection. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1998)*, pages 28–36. ACM, 1998.

[79] Jon M. Kleinberg. Bursty and Hierarchical Structure in Streams. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2002)*, pages 91–101. ACM, 2002.

[80] Jianshu Weng and Bu-Sung Lee. Event Detection in Twitter. In *Proceedings of the 5th International AAAI Conference on Weblogs and Social Media (ICWSM 2011)*. AAAI Press, 2011.

[81] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. In *Proceedings of Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems (NIPS 2001)]*, pages 601–608. MIT Press, 2001.

[82] S. Papadopoulos, C. Zigkolis, Y. Kompatsiaris, and A. Vakali. Cluster-Based Landmark and Event Detection for Tagged Photo Collections. *IEEE MultiMedia*, 18(1):52–63, Jan 2011.

[83] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[84] Konstantin Andreev and Harald Räcke. Balanced Graph Partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124. ACM, 2004.

[85] Chris H. Q. Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D. Simon. A Min-max Cut Algorithm for Graph Partitioning and Data Clustering. In *ICDM*, pages 107–114, 2001.

[86] Jianbo Shi and Jitendra Malik. Normalized Cuts and Image Segmentation. *IEEE TPAMI*, 22(8):888–905, 2000.

[87] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[88] Charles M Fiduccia and Robert M Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181. IEEE, 1982.

[89] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel Static and Dynamic Multi-constraint Graph Partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[90] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. How to partition a billion-node graph. In *ICDE*, pages 568–579, 2014.

[91] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 1222–1230. ACM, 2012.

[92] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining (WSDM)*, pages 333–342. ACM, 2014.

[93] Robert Krauthgamer, Joseph Seffi Naor, and Roy Schwartz. Partitioning Graphs into Balanced Components. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 942–949. Society for Industrial and Applied Mathematics, 2009.

[94] Roger Guimera, Marta Sales-Pardo, and Luis Amaral. Modularity from Fluctuations in Random Graphs and Complex Networks. *Phys. Rev. E*, 70, Aug 2004.

[95] M. E. J. Newman. Finding Community Structure in Networks using the Eigenvectors of Matrices. *Physical Review E - PHYS REV E*, 74:36104, 5 2006.

[96] Pascal Pons and Matthieu Latapy. Computing Communities in Large Networks using Random Walks. In *Computer and Information Sciences-ISCIS 2005*, pages 284–293. Springer, 2005.

[97] Arnaud Browet, Pierre-Antoine Absil, and Paul Van Dooren. Community Detection for Hierarchical Image Segmentation. In *Proceedings of the 14th International Workshop on Combinatorial Image Analysis (IWCIA 2011)*, Lecture Notes in Computer Science, pages 358–371. Springer, 5 2011.

[98] Nurcan Yuruk, Mutlu Mete, Xiaowei Xu, and Thomas A. J. Shweiger. A Divisive Hierarchical Structural Clustering Algorithm for Networks. In *Proc. ICDM Workshops*, pages 441–446, 2007.

[99] Jianbin Huang, Hongbo Deng, Heli Sun, Yizhou Sun, Jiawei Han, and Yaguang Liu. SHRINK: a Structural Clustering Algorithm for Detecting Hierarchical Communities in Networks. In *CIKM*, pages 219–228, 2010.

[100] Heli Sun, Jianbin Huang, Jiawei Han, H. Deng, Peixiang Zhao, and Bo-Qin Feng. gSkeletonClu: Density-Based Network Clustering via Structure-Connected Tree Division or Agglomeration. In *Proc. ICDM*, pages 481–490, Dec 2010.

[101] Jianbin Huang, Heli Sun, Qinbao Song, Hongbo Deng, and Jiawei Han. Revealing Density-Based Clustering Structure from the Core-Connected tree of a network. *IEEE TKDE*, 25(8):1876–1889, 2013.

[102] Zhidan Feng, Xiaowei Xu, Nurcan Yuruk, and Thomas A. J. Schweiger. A Novel Similarity-Based Modularity Function for Graph Partitioning. In *Proc. DaWaK*, pages 385–396, 2007.

[103] Dustin Bortner and Jiawei Han. Progressive Clustering of Networks using Structure-Connected Order of Traversal. In *ICDE*, pages 653–656, 2010.

[104] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. LinkSCAN*: Overlapping Community Detection Using the Link-space Transformation. In *ICDE*, pages 292–303, 2014.

[105] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, pages 226–231, 1996.

[106] J.B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Saleseman Problem. In *Proc. Amer. Math. Soc. 7*, pages 48–50, 1956.

[107] T. S. Evans and R. Lambiotte. Line Graphs, Link Partitions, and Overlapping Communities. *Phys. Rev. E*, 80(1):016105, July 2009.

[108] M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM REVIEW*, 45(2):167–256, 2003.

[109] Matthieu Latapy, Clemence Magnien, and Nathalie Del Vecchio. Basic Notions for the Analysis of Large Two-mode Networks. *Social Networks*, 30(1):31–48, 2008.

[110] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-law Relationships of the Internet Topology. In *Proceedings of the ACM SIGCOMM 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM 1999)*, pages 251–262, New York, NY, USA, 1999. ACM.

[111] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web (WWW 2011)*. ACM Press, 2011.

[112] Andrea Lancichinetti and Santo Fortunato. Community Detection Algorithms: A Comparative Analysis. *Phys. Rev. E*, 80:056117, Nov 2009.

[113] Michael Flynn. Very High-speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[114] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. Designing Fast Architecture-Sensitive Tree Search on Modern Multicore/Many-core Processors. *ACM Transactions on Database Systems (TODS)*, 36(4):22, 2011.

[115] Pawan Harish and PJ Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[116] Thomas Willhalm. Intel® Performance Counter Monitor - A better way to measure CPU utilization | Intel® Developer Zone. `http://www.intel.com/software/pcm`, 2014.

[117] U Kang and Christos Faloutsos. Beyond 'Caveman Communities': Hubs and Spokes for Graph Compression and Mining. In *ICDM*, pages 300–309, 2011.

[118] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *J. ACM*, 46(5):604–632, September 1999.

[119] Baoning Wu and Brian D. Davison. Identifying Link Farm Spam Pages. In *WWW*, pages 820–829, 2005.

[120] Pedro Domingos and Matt Richardson. Mining the Network Value of Customers. In *KDD*, pages 57–66, 2001.

[121] Jianbo Shi and Jitendra Malik. Normalized Cuts and Image Segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, 2000.

[122] Ning Xu, Lei Chen, and Bin Cui. LogGP: A Log-based Dynamic Graph Partitioning Method. *PVLDB*, pages 1917–1928, 2014.

[123] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. Fast Algorithm for Modularity-based Graph Clustering. In *AAAI*, pages 1170–1176, 2013.

[124] Peng Jiang and Mona Singh. SPICi: A Fast Clustering Algorithm for Large Biological Networks. *Bioinformatics*, 26(8):1105–1111, 2010.

[125] Corban G Rivera, Rachit Vakil, and Joel S Bader. NeMo: Network Module Identification in Cytoscape. *BMC bioinformatics*, 11(Suppl 1):S61, 2010.

[126] Yizhou Sun, Charu C. Aggarwal, and Jiawei Han. Relation Strength-Aware Clustering of Heterogeneous Information Networks with Incomplete Attributes. *PVLDB*, 5(5):394–405, 2012.

[127] Jia Wang and James Cheng. Truss Decomposition in Massive Networks. *PVLDB*, 5(9):812–823, 2012.

[128] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. On Triangulation-based Dense Neighborhood Graphs Discovery. *PVLDB*, 4(2):58–68, 2010.

[129] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[130] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM TKDD*, 1(1), March 2007.

[131] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[132] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proc. ICDM*, pages 745–754, 2012.

[133] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[134] Kun-Lin Liu, Wu-Jun Li, and Minyi Guo. Emoticon smoothed language models for twitter sentiment analysis. In *AAAI*, 2012.

Reference Papers

Journal Paper

- Hiroaki Shiokawa, Yasuhiro Fujiwara, Makoto Onizuka, "Fast Extraction Method of Communities in Network Structures," *IEICE TRANSACTIONS on Information and Systems, Vol.J96-D, No.5*, pp.1145-1157, May 2013 (in Japanese with English Abstract).

- Hiroaki Shiokawa, Takeshi Yamamuro, Yasuhiro Fujiwara, Makoto Onizuka, "Parallel Approach for Modularity-based Graph Clusteirng with SIMD Instruction," *DBSJ Journal, Vol.12, No.1*, pp.91-96, Jun 2013 (in Japanese with English Abstract).

Conference Paper

- Hiroaki Shiokawa, Yasuhiro Fujiwara, Makoto Onizuka, "Fast Algorithm for Modularity-based Graph Clustering," *In Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI2013)*, pp.1170-1176, Bellevue, Washigton, USA, July 2013.

Other Papers

Journal Paper

- Junya Arai, Makoto Onizuka, Hiroaki Shiokawa, "Efficient k-Anonymization by Combining Clustering and Space Partitioning," *DBSJ Journal, Vol.13-J, No.1*, pp.72-77, Octorber 2014 (in Japanese with English Abstract).
- Yasuhiro Iida, Yasunari Kishimoto, Yasuhiro Fujiwara, Hiroaki Shiokawa, Makoto Onizuka, "Finding Communities and Ranking for Large-Scale Graphs −Fast Algorithms and Applications−," *Journals of JSAI, Vol.29, No.5*, pp.472-479, September 2014 (in Japanese).
- Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, Makoto Onizuka, "Fast and Exact Personalized PageRank," *IEICE TRANSACTIONS on Information and Systems, Vol.J97-D, No.4*, pp.738-751, April 2014 (in Japanese with English Abstract).
- Hiroaki Shiokawa, Hiroyuki Kitagawa, Hideyuki Kawashima, Yosuke Watanabe, "A High Availability Scheme for Distributed Stream Processing," *IEICE TRANSACTIONS on Information and Systems, Vol.J93-D, No.6*, pp.767-780, Jun 2010 (in Japanese with English Abstract).

Conference Paper

- Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Takeshi Mishima, Makoto Onizuka, "Efficient Ad-hoc Search for Personalized PageRank," *In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD2013)*, pp.445-456, New York, USA, June 2013.

- Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Makoto Onizuka, "Fast and Exact Top-k Algorithm for PageRank," *In Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI2013),* pp.1106-1112, Bellevue, Washington, USA, July 2013.

- Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Makoto Onizuka, "Efficient Search Algorithm for SimRank," *In Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE2013),* pp.589-600, Brisbane, Australia, April 2013.

- Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, Makoto Onizuka, "Efficient Personalized PageRank with Accuracy Assurance," *In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2012),* pp.15-23, Beijing, China, August 2012.

- Hiroaki Shiokawa, Hiroyuki Kitagawa, Hideyuki Kawashima, "A-SAS: An Adaptive High-Availability Scheme for Distributed Stream Processing Systems," *In Proceedings of the 11th International Conference on Mobile Data Management (MDM2010)*, pp.413-418, Missouri, USA, May 2010.